
Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

# Specification of Software Architecture Reconfiguration

Miguel Alexandre Wermelinger

Dissertação apresentada para obtenção do grau de Doutor em Informática, pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Lisboa
(Setembro 1999)

To my daughter, Ana, with the promise of real summer holidays together next year, and to my wife, Claudia, for her ceaseless love and support.

# Acknowledgments

I believe every PhD student needs at least the following people to successfully and sanely complete his dissertation: an effective supervisor, a colleague working in the same research group to discuss technical details with, a colleague working in a different area to provide external and unbiased perspective, and a supporting family. I had the good fortune of counting with excellent people to fulfill these roles.

First and foremost, my supervisor, José Luiz Fiadeiro, from Universidade de Lisboa, gave me guidance when needed but also provided the necessary freedom to pursue my own paths. He taught me much about doing research, provided moral support, was always ready to receive me in his office for a lengthy scientific discussion or for a quick nice chat, produced the necessary bureaucratic paperwork at lightning speed, and arranged the money for my insatiable desire to visit universities and attend conferences in interesting places when all other financial sources had been tried. He is also a wonderful guide to great food and restaurants.

Antónia Lopes, also from Universidade de Lisboa, was always available to discuss and explain in depth the technical details of categories and CommUnity and to answer all my questions—even the stupid ones and those I had already asked but had forgotten the answer. Her patience is truly admirable, especially taking into account how often I went to her office, interrupting her work. She also carefully read a large part of this document, providing numerous suggestions and spotting some embarrassing errors. Her pithy sense of humour makes work more enjoyable. Most of all, I have to thank her the way she—together with José Fiadeiro, Isabel Nunes, and Nuno Barreiro—made me feel welcome in a research group belonging to a university that is not my own.

I have the luck of sharing the office with my dear friend Luís Caires. Thus we easily and frequently engaged in long conversations on reconfiguration, the PhD process in particular and research in general, and many other topics. All this has enriched my professional and personal education more than he may imagine. He also put me up to date with the department's latest gossip after my long retreats at home.

My family provided in abundance the necessary relaxation but also the support that allowed me to dedicate exclusively to work when needed. I owe them all more than I can ever express.

I am indebted to Daniel Le Métayer, Narciso Martí-Oliet, and some anonymous reviewers for their helpful comments on drafts of papers which helped to improve the presentation. I also thank Andrea Corradini and Manuel Koch for answering some questions on graph grammars.

I gratefully acknowledge the financial support of the following institutions: Association for Computing Machinery; Conselho de Reitores das Universidades Portuguesas; Departamento de Informática e Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa; Fundação Calouste Gulbenkian; Fundação para a Ciência e Tecnologia through projects PRAXIS XXI 2/2.1/MAT/46/94 (ESCOLA—Executable and Verifiable Specifications of Concurrent Systems: Languages and Models), PRAXIS XXI PCEX/P/MAT/46/96 (ACL—Algebraic Combination of Logics), and PRAXIS XXI 2/2.1/TIT/1662/95 (SARA—Societies of Animated and Responsible Agents); Fundação Luso-

Americana para o Desenvolvimento; Fundação Oriente; Laboratório de Modelos e Arquitecturas Computacionais; Reitoria da Universidade Nova de Lisboa; ESPRIT network of excellence RENOIR (Requirements Engineering Network Of International cooperating Research groups).

Finally, I would like to manifest my appreciation for the many people that developed the free software used to produce this document.

# Sumário

Nos últimos anos, as arquitecturas de software têm recebido crescente atenção por parte das comunidades académica e industrial como meio de estruturar o desenho de sistemas complexos. Uma das áreas de interesse é a possibilidade de reconfigurar arquitecturas para permitir a adaptação dos sistemas por elas descritos a novos requisitos. A reconfiguração consiste em adicionar ou remover componentes ou ligações e pode ter de ocorrer sem parar a execução do sistema a alterar. Este trabalho contribui para uma descrição formal desse processo.

Partindo do princípio que raramente um único formalismo consegue satisfazer plenamente todos os requisitos em todas as situações, apresentam-se três abordagens, cada uma com diferentes pressupostos sobre os sistemas a que se aplicam e com diferentes vantagens e desvantagens. Cada uma tem como ponto de partida trabalho feito por outros investigadores e tem a preocupação estética de alterar o menos possível o formalismo original, mantendo o seu espírito.

A primeira abordagem mostra como uma dada reconfiguração pode ser especificada da mesma forma que o sistema ao qual se aplica e de modo a ser executada da maneira mais eficiente possível. A segunda abordagem explora a "Chemical Abstract Machine", um formalismo de reescrita de multiconjuntos de termos, para uma descrição uniforme de arquitecturas, computações e reconfigurações. A última abordagem usa uma linguagem de desenho de programas paralelos similar ao UNITY para descrever computações, representa as arquitecturas por diagramas no sentido da Teoria das Categorias, e especifica a reconfiguração por regras de transformação de grafos.

# Abstract

In the past years, Software Architecture has attracted increased attention by academia and industry as the unifying concept to structure the design of complex systems. One particular research area deals with the possibility of reconfiguring architectures to adapt the systems they describe to new requirements. Reconfiguration amounts to adding and removing components and connections, and may have to occur without stopping the execution of the system being reconfigured. This work contributes to the formal description of such a process.

Taking as a premise that a single formalism hardly ever satisfies all requirements in every situation, we present three approaches, each one with its own assumptions about the systems it can be applied to and with different advantages and disadvantages. Each approach is based on work of other researchers and has the aesthetic concern of changing as little as possible the original formalism, keeping its spirit.

The first approach shows how a given reconfiguration can be specified in the same manner as the system it is applied to and in a way to be efficiently executed. The second approach explores the Chemical Abstract Machine, a formalism for rewriting multisets of terms, to describe architectures, computations, and reconfigurations in a uniform way. The last approach uses a UNITY-like parallel programming design language to describe computations, represents architectures by diagrams in the sense of Category Theory, and specifies reconfigurations by graph transformation rules.

# List of Symbols

Within each group, symbols are listed alphabetically, special characters coming first. Some symbols have different meanings, depending on the context.

## Sets and Functions

| | |
|---|---|
| $\emptyset$ | empty set |
| $\lvert A \rvert$ | cardinality of set $A$ |
| $\setminus$ | set difference |
| $\uplus$ | disjoint union of sets |
| $f : A \rightharpoonup B$ | partial function from $A$ to $B$ |
| $f; g$ | function composition: $g(f(x))$ |
| $+_n$ | addition modulo natural number $n$ |
| $\mathbb{N}$ | natural numbers |
| $\pi_i$ | projection on the $i$-th element of a tuple |

## Transaction Approach

| | | |
|---|---|---|
| $<$ | command order | Definition 2.6 on page 20 |
| $/$ | transaction dependency relation | Definition 2.3 on page 17 |
| $C$ | reconfiguration commands | Definition 2.6 on page 20 |
| $D$, $D_n$ | port dependency relation (of node interface $n$) | Definition 2.1 on page 16 |
| $I$, $I_n$ | initiator ports (of node interface $n$) | Definition 2.1 on page 16 |
| $N$ | node interfaces | Definition 2.2 on page 17 |
| $R$, $R_n$ | recipient ports (of node interface $n$) | Definition 2.1 on page 16 |
| $T$ | transactions | Definition 2.2 on page 17 |

## Grammars

| | |
|---|---|
| ‘x’ | terminal symbol |
| *x* | non-terminal symbol |
| [. . . ] | optional |
| . . . $^*$ | sequence of zero or more |
| . . . $^+$ | sequence of one or more |
| . . . \| . . . | alternative |
| (. . . ) | grouping |

## CHAM

| | |
|---|---|
| $\triangleleft$ | airlock |
| $\{\!\!\{\}\!\!\}$ | membrane |
| $\ldots \rightarrow \ldots$ | reaction rule |

## Graphs

| | | |
|---|---|---|
| $\emptyset$ | empty graph | Example A.13 on page 117 |
| $A$ | arcs | Definition A.1 on page 109 |
| $G_i$ | paths of length $i$ in graph $G$ | Notation A.4 on page 109 |
| $L_X$ | labels for nodes or arcs $X$ | Definition A.9 on page 110 |
| $\mathrm{lbl}_X$ | labelling function for nodes or arcs $X$ | Definition A.9 on page 110 |
| $N$ | nodes | Definition A.1 on page 109 |
| $\mathrm{src}$ | source function | Definition A.1 on page 109 |
| $T$ | transitions of a labelled transition system | Notation A.12 on page 110 |
| $\mathrm{trg}$ | target function | Definition A.1 on page 109 |
| $W$ | worlds of a labelled transition system | Notation A.12 on page 110 |
| $w_0$ | initial world of a labelled transition system | Notation A.12 on page 110 |

## Category Theory

| | | |
|---|---|---|
| $f; g$ | morphism composition | Definition A.13 on page 111 |
| $\mathfrak{C}$ | category | Definition A.13 on page 111 |
| $\lvert\mathfrak{C}\rvert$ | objects of $\mathfrak{C}$ | Definition A.13 on page 111 |
| $\langle \Delta_D, \delta_D \rangle$ | diagram D with graph $\Delta_D$ and labelling $\delta_D$ | Notation A.18 on page 112 |
| $G_{\mathfrak{C}}$ | graph of category $\mathfrak{C}$ | Definition A.13 on page 111 |
| $\mathrm{Hom}_{\mathfrak{C}}(x, y)$ | morphisms of $\mathfrak{C}$ from $x$ to $y$ | Notation A.14 on page 111 |
| $\mathrm{id}(x)$ | identity morphism for object $x$ | Definition A.13 on page 111 |
| $(\mathfrak{C} \downarrow x)$ | comma category | Definition A.16 on page 111 |

## Graph Grammars

| | | |
|---|---|---|
| $G \overset{p,m}{\Longrightarrow} H$ | direct derivation with production $p$ and match $m$ | Definition A.32 on page 116 |
| $K$ | interface of graph production | Definition A.31 on page 116 |
| $L$ | left-hand side of graph production | Definition A.31 on page 116 |
| $R$ | right-hand side of graph production | Definition A.31 on page 116 |

## CommUnity

| | | |
|---|---|---|
| $\bot$ | idle action | Definition 4.14 on page 53 |
| $A$ | actions | Definition 4.14 on page 53 |
| $\beta$ | program body | Definition 4.23 on page 61 |
| $C$ | channel | Definition 4.42 on page 73 |
| $D(x)$ | domain of variable or action $x$ | Notation 4.15 on page 54 |
| $E(a,o)$ | right-hand side of assignment to $o$ in action $a$ | Definition 4.23 on page 61 |
| $\epsilon$ | environment of a program instance | Definition 4.34 on page 69 |
| $F$ | pre-defined functions | Definition 4.1 on page 50 |
| $G$ | glue of a connector | Definition 4.42 on page 73 |
| $G(a)$ | guard of action $a$ | Definition 4.23 on page 61 |
| $\gamma$ | glue morphism | Definition 4.42 on page 73 |
| $I$ | input variables | Definition 4.14 on page 53 |
| $ic$ | initialisation condition | Definition 4.23 on page 61 |
| $\mathcal{IP}$ | functor from program instances to programs | Proposition 4.12 on page 71 |
| $LV$ | logical variables | Definition 4.34 on page 69 |
| $O$ | output variables | Definition 4.14 on page 53 |
| $\psi$ | program signature | Definition 4.20 on page 55 |
| $R$ | role | Definition 4.42 on page 73 |
| $\rho$ | role morphism | Definition 4.42 on page 73 |
| $S$ | predefined sorts | Definition 4.1 on page 50 |
| $U$ | track length | Section 4.1 on page 49 |
| $V$ | program variables | Notation 4.15 on page 54 |
| $Vars(D)$ | logical variables used in diagram $D$ | Notation 4.39 on page 71 |
| $\mathcal{V}_w$ | valuation at world $w$ | Notation 4.18 on page 54 |

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

Most software systems must undergo several modifications during their lifetime in order to cope with new human needs (i.e., new requirements), new technology (e.g., new implementation), or a new environment (e.g., if a hardware component fails or a new one is added). For economical or safety reasons (e.g., in banking), some systems cannot be stopped or taken off-line to perform those changes: they have to be dynamically reconfigured. Even if it is possible to stop the application to change it, it may be more advantageous to reconfigure it at run-time because unchanged sections can continue to provide partial service.

Changes are not always imposed by some external entity like the user or the system designer. Due to the technological advances in the past years—of which cellular phones, the World Wide Web, and Java's run-time component loading capabilities are striking examples—and the transient nature of interactions they support, many software systems have an intrinsically dynamic configuration.

The problem of (dynamic) reconfiguration may involve all levels (from the operating system to the application) and all phases (requirements to implementation) of software development. It also requires solutions to many technological problems. In this dissertation we restrict ourselves to a problem that has received relatively little attention and yet is fundamental to use reconfiguration facilities in an easier and more systematic way: how to allow the designer to formally specify the reconfiguration process at a high level of abstraction, namely the application's architecture.

## 1.2 Context

Software Architecture (SA) [PW92, SG96a, SG96b, Per97] is the discipline that addresses the high-level structure of systems, providing the framework in which to satisfy requirements and serving as a basis for the ensuing design phase. The description of the architecture of a software system basically states the components it is made of, how they interact, and what quantitative and qualitative constraints components and interactions must satisfy (e.g., throughput, security, conformance to standards, etc). A software architecture is useful to promote reuse (e.g., for product lines), to help manage the software process, and to choose among design alternatives.

Three of the most important concepts put forward by research in SA are Architecture Description Languages (ADLs), connectors, and styles. Languages provide a precise specification of the architecture. To describe complex systems, architectures may be *hierarchic*, i.e., a component may be specified by a sub-architecture. Normally there

is a supporting toolset that facilitates the construction of architectures from existing components and connectors and provides some (semi-)automatic analysis capabilities. A detailed comparative survey on several ADLs is [Med97]. Connectors are first-class entities to express complex interactions between system components, thus facilitating the separation of coordination from computation. Styles capture sets of restrictions on components and their connections leading to patterns that occur often in software systems (client-server, pipe-filter, etc.).

It has been long recognised that software architectures could be helpful regarding the evolution of systems during their life-time. On the one hand, they might be used to check whether the implementation has drifted from the reference architecture; on the other hand, the high-level synthetic view provided by an architectural description might make more apparent which parts of the system are amenable to change and which changes are desirable and possible. In recent years the need for mechanisms to express changes to architectural descriptions has been manifested in several ways: by participants in conferences [Wol97], by researchers surveying the field [SG94, Med97, Per97], by the existence of an Architecture and Generation Technology Cluster in the DARPA-sponsored Evolutionary Design of Complex Software programme [edc97], and by a growing number of papers on the subject (see Section 1.4 on the next page). This has culminated in the explicit recognition of "dynamic architectures" as a major topic within SA through a dedicated track in the last International Software Architecture Workshop [MP98].

Being a young discipline, the state-of-the-art is still far from the full promise of SA, and there is still no agreement on most issues, concepts, and terminology. The notable exception is some consensus on the structural properties of architectures, which lead to the development of the architecture interchange language ACME [GMW97].

## 1.3 Issues

There are several issues involved in reconfiguration, as discussed in [KM85, KM90, Ore96, Wer98c, MG99]. We let aside technological issues (like the necessary support from the operating system and the component programming language), and classify and summarise the remaining ones as follows.

**time** Architectures may change

1. before compilation,
2. before execution, or
3. at run-time.

The third case is usually called *dynamic reconfiguration*, but some authors (like the research group from the University of California at Irvine [Ore96, Med97]) use the term *dynamic architectures*. We prefer the former since the latter has also been used to refer to architectural descriptions that emphasize behaviour [SG96a]. Each of the three cases requires different supporting technology, but for our work we are only interested whether changes are executed off-line, when the system is shut down, or on-line, while it is running, because that affects whether application state has to be taken into account or not. We therefore do not distinguish the first two cases.

**source** Changes may be triggered by the current state of components or topology of the architecture. This is called *programmed* [End94, MG99] or *constrained run-time* [Ore96] modification. Reconfigurations may also be asked for by the user. This is called *ad-hoc* [End94], *evolutionary* [MG99], or simply *runtime* [Ore96] change. We

adopt the terminology of [End94]. Programmed reconfigurations are described by "change scripts" together with conditions when they are to be executed. Scripts are usually given with the initial architecture, while ad-hoc reconfigurations are requested unpredictably.

**operations** The four fundamental *reconfiguration commands* provided by almost all languages and systems are addition and removal of components and connections. Their names vary, of course: for example, `link` [KM90], `weld` [Ore96], `attach` [ADG98] all create connections. By the very definition, programmed reconfiguration requires operations that query at run-time relevant properties of the system, be they of computational (e.g., the state of components), structural (e.g., the topology of the architecture), or other nature (e.g., the version of a component to be upgraded). Such operations also allow writing general, reusable change scripts. For example, it becomes possible to remove any component satisfying some topological property (e.g., number of connections) because the script may query its identifier instead of relying on some fixed name. This is also useful because a programmed reconfiguration script, although given with the initial architecture, may be executed when the architecture has already been changed (by another script or directly by the user). The query operations thus assess the current architecture, and provide the actual components and connections to be used as arguments of the reconfiguration commands in the script.

**constraints** Changes must preserve the consistency of the system. Moazami-Goudarzi [MG99] distinguishes three cases: structural integrity (e.g., the architecture must keep a ring topology), mutually consistent component state (e.g., a client must not be removed if the server is still processing its request), and application state invariants (e.g., exactly one component holds the token in a ring topology). The second and third cases are only relevant for dynamic reconfiguration. We should add that any kind of property (functional, behavioural, etc.) may serve as a constraint if it is to be preserved by change. Programmed reconfiguration may automatically enforce certain constraints by using their negation as triggering conditions for executing scripts that take corrective action.

**specification** In the limit, a specification is written in three languages: the architecture is described using an ADL, the reconfigurations using an Architecture Modification Language (AML), and the restrictions with an Architecture Constraint Language (ACL). All of them should be declarative, understandable, and analysable. In particular, the changes should be verifiable against the constraints. They also should be modular to allow compositional specifications.

**management** The reconfiguration process may be managed in an explicit and centralized manner by a *configuration manager* [KM85] (also called *architecture evolution manager* [OT98], *coordinator* [Mét98] or *configuror* [ADG98]), or management is implicit and distributed among the components. The latter case has been called *self-organising architectures* [MK96b]. The configuration manager translates the AML specification into low-level commands of the underlying platform. In case of dynamic reconfiguration, the manager should minimise the disruption caused to the running system. The manager should be general-purpose, i.e., not tailored to any specific application domain.

## 1.4 Related Work

Most of the existing work on reconfiguration stems from the Distributed Systems community (see, e.g., [CDS96, CDS98]). As to be expected, it is mainly concerned with

the technology needed to enable reconfiguration. As a result the approaches are often suited to a specific application or domain; they are based on particular frameworks (e.g., CORBA) or programming language (e.g., Java [MG99]); they only deal with certain classes of systems (e.g., client-server [Kin93]) or reconfiguration (e.g., replacement by a copy [HP93]).

Many of the current approaches follow the Configuration Programming philosophy of Kramer and colleagues. Its principles are briefly stated in [KM98]:

1. The configuration language used for structural description should be separate from the programming language used for basic component programming and from the specification language used for specifying component behaviour.
2. Components should be defined as context independent types with well-defined interfaces and should specify the visible behaviour at the component interface.
3. Using the configuration language, complex components should be definable as a composition of instances of component types, and complex specifications should be the composition of component specifications.
4. Change should be expressed at the configuration level, as changes of the component instances and/or their interconnections and of component specifications and/or their interaction.

A detailed survey on many of the existing approaches with a discussion of their merits and drawbacks can be found in [MG99].

Dynamic reconfiguration also occurs in Mobile Computing. Due to change of locations, computational agents may appear and disappear within some system boundary, and the interaction patterns between them vary. Probably the most prominent formalism for mobility is the $\pi$-calculus [Mil99], a process algebra. A state-based formalism is MOBILE UNITY [MR98, RMP97], which extends the parallel program design language UNITY [CM88] and its associated proof logic in order to provide useful programming abstractions to describe transient interactions among programs. The description of a system has a single, separate section that contains all statements necessary to describe interactions. Such statements are able to change the state of the programs (e.g., to transmit a value from one to another). They may also be reactive, i.e., their execution is triggered when a condition on the system state becomes true. This allows to guarantee consistency in face of change.

PoliS [CFM98] and MobiS [Mas99] are formal approaches for code mobility based on Linda-like tuple spaces and the chemical computation model. Tuples may represent data or rules, and spaces may be nested. A rule specifies a tuple rewriting step. Tuples may be consumed and produced in the same space as the rule or in the parent space. This allows data and code to move along the space tree. The difference between the two languages is that PoliS comes equipped with a temporal logic to express properties and a model-checker, while MobiS makes spaces first-class entities, by representing them as tuples, allowing them to move also.

Due to the different foci of mobile and distributed systems, many of the approaches do not capture architectural abstractions, like hierarchic decomposition and connectors. Others are not at the architectural level: they only deal with the implementation, or do not show explicitly which components are allowed to interact and in which ways. Hence we have to turn to work done in SA.

Only few ADLs are able to express dynamism [Med97]. Darwin [MK96a] only permits constrained dynamism: the initial architecture may depend on some parameters, and during run-time components may be replicated. To show the interaction between reconfiguration and the ongoing computations a different formalism, Finite State Processes, is used. Its semantics is given by labelled transition systems and model-checking tools are used to check safety and liveness properties [MKG99].

The C2 language does not have any reconfiguration capabilities by itself; they are provided by a separate AML [Med96]. The ACL used to enforce constraints on reconfiguration is Armani [Mon98], which allows to write propositions in a subset of first-order logic with primitive predicates to query the architecture's topology.

Rapide [LV95] is intended to model architectures of concurrent hardware and software systems and to allow the simulation of their execution. Components' behaviour is described as a set of events partially ordered by time or causality. Event patterns describe succinctly a partially ordered set of events. Patterns are used to impose constraints on the allowed behaviours. Moreover, rules can use patterns to generate new events based on the occurrence of other. The language only allows changing the communication topology of an architecture. However, recent work [VPL99] added primitive events corresponding to the four main reconfiguration operations. As architectures may be hierarchic (i.e., components are organised into a tree), the authors also added an event to change the parent of a component. Patterns may be used to restrain the possible architectural changes, and such constraints may be functionally dependent since behaviour is also represented by events.

Wright [ADG98] uses a slight variation of the process algebra CSP to describe component behaviour. Reconfigurations are specified also in CSP, using primitive actions to add and delete components and links. The semantics provides a translation into pure CSP but it is a bit cumbersome; it requires all distinct configurations to be uniquely tagged because CSP, unlike the $\pi$-calculus, only allows static configurations. Properties are verified with a model-checking tool.

ACME's proposal only allows for the specification of optional elements (i.e., components, connectors, and links) [MGW97].

LEDA [CPT99] uses the $\pi$-calculus to specify the behaviour of components. There are two primitive operators to create new components and to create connections depending on some condition. Due to the name passing of the $\pi$-calculus, it is possible to dynamically establish communication channels that are not explicitly captured at the structural description level of LEDA.

There has been also formal work that is not related to any ADL. Le Métayer [Mét98] describes architectures as graphs, reconfigurations are specified by graph rewrite rules, and computations by a specially designed language, inspired on CSP.

Taentzer et al. [TGM98] have the most uniform framework we are aware of. They represent the architecture as well as the state of components by graphs. Reconfigurations and computations are thus described by graph rewrite rules. Components may export and import part of their state, i.e., graphs, and hence graph rewrite rules also describe communication. The authors impose many constraints on the form of rewrite rules due to their three-fold use.

We [FWM99] represent architectures also as graphs, but nodes (i.e., components) are labelled with name/value pairs to indicate their state. The allowed reconfigurations, creation and removal of connectors, only cater for transient interactions. Graphs are encoded as terms, and reconfiguration and computation are then described using Rewriting Logic [Mes96]. The rules for reconfiguration rewrite the graph while those for computation—one for each action of each component—rewrite the labels. If a connector synchronises two actions of different components, the corresponding rewrite rules must be executed simultaneously to guarantee the correct semantics. But if the connector is removed, they do not have to any more. The rewriting strategy must therefore be dynamically changed.

Hirsch et al. [HIM99] use labelled hypergraphs (i.e., graphs where arcs may connect more than two nodes) to represent architectures. Contrary to other approaches, components are represented by hyperarcs and connections by nodes. The label of a component indicates its current state and the label of a connection represents a communication action. A specification has separate rewrite rules for computations/interactions

and reconfigurations. Computations are thus described as label rewriting. All rules are context-free: the left-hand side is always a single hyperarc with the nodes it connects. Rules may not delete any nodes, because other arcs may be linked to them. Moreover, as the authors remark, a simple reconfiguration like adding a connection between two existing components is not possible. When communication occurs, all rules corresponding to the involved components must be applied simultaneously. For this purpose, the authors use constraint-solving (where the constraints are equalities of node labels) to obtain a set of context-dependent rules from the context-free rules given by the user. There are only two kinds of communications (and hence two types of nodes): point-to-point and broadcast. The former is easier since it amounts to combine the communication rule of the sender with the one of the receiver.

To sum up, the existing approaches, taken collectively, have the following drawbacks:

- arbitrary reconfigurations are not possible;
- they are not at the architectural level;
- computations are described with simple, low-level languages which do not capture some of the abstractions used by programmers and hence may lead to cumbersome specifications;
- the interaction between computation and reconfiguration—needed for dynamic reconfiguration—either leads to additional, possibly complex, linguistic or semantic constructs, or is not explicit, or not cleanly separated.

## 1.5 Our Approaches

Our goal is to be able to formally specify arbitrary dynamic reconfigurations at the architectural level in a simple, explicit, and adequate way. Given the diversity of problems that need to be addressed, it is futile to search for a "universal" formalism. Hence we present three different approaches, each addressing different problems and emphasizing different aspects. We adopt the Configuration Programming philosophy, giving more importance to separation of concerns (reconfiguration vs. computation) than to separate languages. We do not invent new languages or formalisms, and changes to existing ones are kept simple, small, and in the spirit of the original. The common aspect of the three approaches is the use of graphs due to their suitability for architectures, their simple but rigorous mathematical basis, and their intuitive depiction capabilities.

The first approach, based on work by Kramer and colleagues [KM90, GK96], deals only with the specification of the efficient and modular execution by the configuration manager of a given set of reconfiguration commands. Its major contributions with respect to the original work are: to minimise the disruption time; to handle hierarchic architectures; to represent reconfiguration processes and the systems they are applied to in the same way.

The other two approaches are used to describe the allowed reconfigurations and their effects. Both are based on rewriting in order to have at all times an explicit architectural model. As a by-product, they can generate the reconfiguration commands to be processed by the first approach. The second approach, inspired by work done by Le Métayer [Mét98] and Inverardi and Wolf [IW95], explores the Chemical Abstract Machine (CHAM), a formalism for rewriting multisets of terms, to describe architectures, computations, and reconfigurations in a uniform way. The original work by Inverardi and Wolf on the application of the CHAM to Software Architecture only deals with systems whose architecture does not change and is specified in a monolithic way. The main contribution of our approach is the definition of a simple CHAM-based ADL and methodology

to define dynamically reconfigurable hierarchic software architectures from reusable component specifications.

Finally, we extend the categorical approach to parallel program design, initially developed by Fiadeiro and Maibaum [FM97], in order to handle dynamic reconfiguration. It uses a UNITY-like language to describe computations and represents architectures by diagrams in the sense of Category Theory. To specify reconfigurations, we add graph productions as defined by Ehrig and colleagues [CMR+96a]. The main contribution of our approach is to provide a uniform algebraic framework for dynamic reconfiguration. The framework also provides a simple notion of style and the automatic maintenance of a style during reconfiguration, and operations to construct new connectors from existing ones. Moreover, the combination of the categorical framework with a program design language has none of the problems listed at the end of the previous section.

Each approach is presented in a separate, self-contained chapter with its own abstract and conclusions. The appendix contains the mathematical definitions needed for Chapter 4. Some notation (like the syntax of context-free grammars) is only defined in the List of Symbols on page viii. Much of the material in this dissertation has been presented previously in a preliminary form [Wer97, WF98b, WF98a, Wer98b, Wer98c, Wer98a, WF99, Wer99].

# Chapter 2

# The Transaction Approach

In this chapter we adopt a framework developed by Kramer and colleagues [KM85, KM90, GK96]. It is simple and general, both in terms of the changes it allows and in terms of the assumptions it makes on systems. Changes must occur in a consistent state, which is brought about by "freezing" some system components. Upon closer analysis of the two algorithms proposed for computing the set of those components, we find that neither is minimal regarding the disruption it causes to the system. Switching to a connection based approach we come up with a conceptually very simple yet effective minimal solution.

However, that only accounts for disruption in terms of "size", i.e., what parts of the system are "frozen". It does not take into account for how long they are inactive. Since we work with an abstract, implementation-independent reconfiguration model, our solution just provides an execution order for the change commands such that they are performed as much in parallel as the logical dependencies between them allow.

The third contribution of this chapter is the treatment of hierarchic systems, whose components can be made of interconnected subcomponents. For practical purposes, the original work only deals with flat systems. The hierarchic reconfiguration management method to be introduced allows the parallel execution of change commands in different subsystems while taking into account any dependencies among them. Furthermore the method is as modular as the system it is applied to.

This chapter is a slightly revised and extended version of [Wer97].

## 2.1 The Original Model

We adopt the reconfiguration model developed in [KM85, KM90] and summarised in Figure 2.1 on the next page[1]. In the following we describe the assumptions made by the model for each element appearing in the diagram.

A system can be depicted as a directed graph whose nodes are the system components and whose arcs are connections between components. The model assumes there is at most one connection between any pair of components. A transaction is a sequence of one or more message exchanges along a connection. An arc from a node N to a node N′ states that all transactions along that connection are initiated by N, although during a transaction communication flow can occur in both directions. Transactions complete in bounded time and the initiator is always informed of completion. In particular, the system does not get into any deadlock or livelock situation. These assumptions help to prove that the consistent state can be reached in finite time and that the configuration manager knows when. A transaction t is *dependent* on the *consequent* transactions

[1] Figures 2.1 to 2.5 are adapted from [KM85, KM90, GK96].

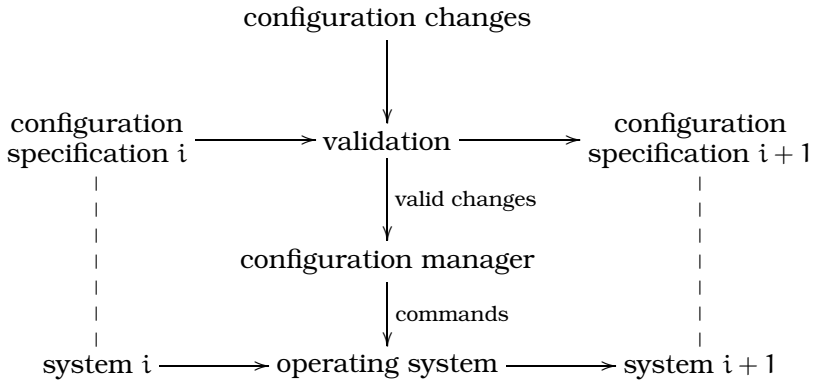



Figure 2.1: The dynamic reconfiguration model

$t_1, t_2, \ldots$ (written $t/t_1 t_2 \ldots$), if its completion depends on the completion of all the other ones. Otherwise a transaction is called *independent*.

Changes to a system are specified using four commands, to be executed by the operating system, with obvious meanings: `create` N, `remove` N, `link` N `to` N′, `unlink` N `from` N′. Given a specification of the current system configuration and the specification of the configuration changes, the validation process checks whether the changes may be (totally or partially) applied to the system and produces the specification of the resulting system. Checks may range from simple syntactic ones (e.g., `remove` N is incorrect if N does not exist in the system) to deep semantic results (e.g., is the resulting system deadlock free?). In the following it is assumed that changes are valid and that the specification is declarative, i.e., the change commands are not in any particular order.

Given the valid changes, the configuration manager generates the instructions for the operating system to reconfigure the current system, such that the resulting one conforms to the specification produced by the validation process. In particular, the manager performs the following steps:

1. Compute from the change specification the nodes that must be in a consistent state for reconfiguration to take place.

2. Compute the nodes that must become "frozen" in order to achieve consistency over the set of nodes obtained in the previous step.

3. Send a "freeze" message to each node obtained in step 2 and wait for all the acknowledgments.

4. Instruct the operating system to execute changes in the following order: `unlink`, `remove`, `create`, `link`.

5. Instruct the created and the "frozen" nodes (except the removed ones) to resume processing.

There are two approaches based on this model that differ only in steps 2 and 3. The first one [KM90], which we call the passive approach, "freezes" a node by preventing it from initiating any new transaction; the second one [GK96] completely stops the node's execution and therefore we call it the blocking approach.

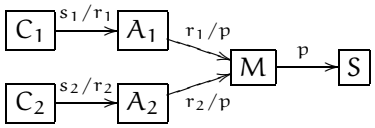



Figure 2.2: A client-server system with dependent transactions

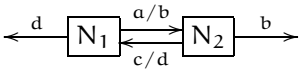



Figure 2.3: Mutual dependencies

### 2.1.1 The Passive Approach

In this method [KM90] the "frozen" state is called *passive* and the "freeze" message is `passivate`. To facilitate exposition, let us first handle only independent transactions.

A component is passive if it is not engaged in transactions it initiated and if it will not start new ones. However, it must accept and service transactions in order to let other nodes become passive. Therefore, passiveness is reachable in finite time: a component just has to wait for the transactions it initiated to finish (this is guaranteed to happen) and then make sure it will not start new ones. The passive state is just a necessary condition for reconfiguration. In order to guarantee a consistent and stable internal state, in addition to being passive a node should not have any outstanding transaction to service. This is called *quiescence* and depends on those components that can initiate transactions with the node. Therefore, the *passive set* of a node $Q$, $PS(Q)$, is defined as $Q$ and all nodes with connection arcs towards $Q$. It is easy to see that $Q$ is quiescent if all nodes in $PS(Q)$ are passive.

The *quiescent set* $QS$ for a given change specification is the set of nodes that must be quiescent during the reconfiguration, namely those that will be removed and the initiators of transactions that will be added or removed. Newly created nodes are automatically quiescent. The set of nodes to "freeze", called change passive set, is then simply $CPS = \bigcup_{i \in QS} PS(i)$.

To see why this does not work for dependent transactions, consider a system with clients $C_i$ accessing through agents $A_i$ a server $S$ managed by $M$ (Figure 2.2). If the server is going to be replaced, then both $S$ and $p$ will be removed. Thus the configuration manager calculates $QS = \{M, S\}$ and $CPS = \{A_1, A_2, M, S\}$. However, if a client has a new request $s_i$, then the respective agent cannot service it because according to the definition of passiveness it may not initiate $r_i$ (on which $s_i$ depends). This would lead to a partially incomplete transaction, i.e., to an inconsistent state of the whole system during reconfiguration. On the other hand, allowing $A_i$ to start transaction $r_i$ would lead to new transactions on the manager and on the server, which therefore would not be in the quiescent state.

Another problem is that reachability of the quiescent state in bounded time is lost. For example, if $A_1$ is to be replaced, then $QS = CPS = \{C_1, A_1\}$. If $A_1$ becomes passive before $C_1$, and $C_1$ just initiates a new transaction $s_1$ before getting the `passivate` command from the configuration manager, then the client will never become passive because $r_1$ is not initiated. In this case one could order the commands (`passivate` $C_1$ before `passivate` $A_1$), but for systems with mutual dependencies like the one in Figure 2.3 no such ordering is possible.

To solve these problems, the notion of passive state is changed. The characterisation

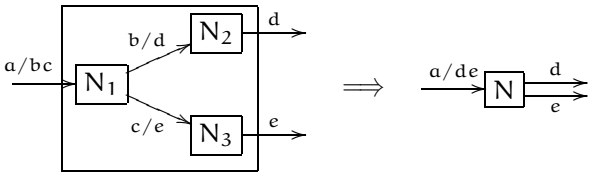



Figure 2.4: Composing dependencies

of passive set must also change, since the nodes that may initiate transactions with a given node are not just its immediate neighbours. The new definitions are thus as follows.

In the *generalised passive state* a node is not engaged in non-consequent transactions it initiated and it will not initiate new ones. Furthermore the node accepts and services all requests, initiating consequent transactions if necessary. The *enlarged passive set* of a node Q, $\mathrm{EPS}(Q)$, includes Q and all nodes that can initiate transactions which result in consequent transactions on Q.

Notice that both definitions reduce to the old ones in case all transactions are independent. The reconfiguration algorithm remains the same, except that $\mathrm{PS}(i)$ is substituted by $\mathrm{EPS}(i)$ in the calculation of CPS.

The server replacement in Figure 2.2 on the preceding page is now correctly handled. Since $\mathrm{EPS}(S) = \{C_1, A_1, C_2, A_2, M, S\}$, all nodes have to be passivated. Even if all components but $C_1$ are already passive, any pending $s_1$ transaction will be serviced (through $A_1$ and M) by the server and therefore the client can become passive and reconfiguration may start.

In general, systems are not flat as assumed until now but hierarchic, i.e., some nodes (called *composite*) are made of connected subnodes. A composite node is connected to other nodes through some of its subcomponents. The transaction dependency of a composite component must be derived from its subcomponents. The following substitution rule is given in [KM90]:

> "in composing two nodes, substitute the consequents for each occurrence of the dependent transaction which is hidden by the composition."

The rule can be iterated on components and connections (Figure 2.4). To simplify reconfiguration management, [KM90] suggests that a composite node is considered to be passive if all its subnodes are, and that all transactions between composite nodes are independent.

### 2.1.2 The Blocking Approach

An alternative method is presented in [GK96]. It assumes that a node is consistent and self-contained except during transactions, as those are the only interactions with the outside environment. Thus, to make a node quiescent it is enough to block it while it is idle (not engaged in any transaction). A component is also assumed not to interleave transactions: while handling a request a node may not service any new one, even if it comes from a different connection, and it may initiate only consequent transactions. This is used to prove that the blocked state is reachable in finite time.

The basic algorithm is thus to send a `block` message to the nodes in the quiescent set (called BSet, short for *blocking set*, in [GK96]). As soon as such a node N is idle, it blocks and sends an acknowledge to the configuration manager. Since some of the nodes that depend on N may also have to block, N must temporarily unblock to service some requests. However, it must be guaranteed that at some point no more such requests will

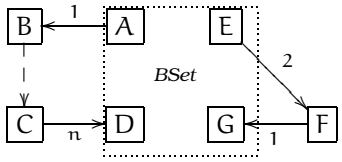



Figure 2.5: A blocking problem

arrive and N will remain blocked. The basic question is therefore: what transactions should a blocked node service?

It is obvious that it cannot process just any incoming transaction, since it might come from a node that is not affected in any way by the reconfiguration and as such might initiate a new transaction any time. Thus the blocked node would have to unblock unpredictably and the safe state needed for reconfiguration to begin would never be reached. It is also evident that at least the transactions initiated by other BSet members will have to be serviced in order for them to become blocked. On the other hand, not every request from a non-BSet member can be ignored. Consider the cases depicted in Figure 2.5. Node D must service the request from C because it is the $n$-th consequent transaction of a transaction initiated by A, which must be completed for A to become blocked. In the second case on the right half of the figure, component F has initiated a transaction with G before getting a request from E. If G does not service the transaction, F will never be able to start attending E's request since transactions do not interleave.

One could let BSet nodes unblock just in those situations but the authors argue this is non-trivial and has great run-time overhead. Instead they propose the BSet to grow dynamically in step with outgoing transactions. When a node gets a request from a BSet member, it becomes a member too, and only requests from BSet members are attended; all other are queued and serviced after the reconfiguration. In the previous cases, it means that the BSet grows to encompass the whole system, and therefore D and G will service the transactions initiated by C and F respectively.

Notice that the BSet has two kinds of members: those that "really" must block due to the reconfiguration and those that block in order to let members of the first kind to get blocked. Therefore a distinction is made between the original BSet and the extended BSet. Their union is the BSet. When all the original BSet nodes become blocked, the components in the extended BSet can be unblocked. The disruption thus first grows and then shrinks.

As the calculation of the BSet is dynamic, the reconfiguration algorithm is distributed through the configuration manager and the nodes. Each node runs the same code in a transparent way using hooks that are called upon relevant events like message arrival (from another node or from the configuration manager), transaction begin and transaction end. The application programmer only has to mark in the component's code where the last two hooks have to be activated. The code run by each hook basically updates local variables and sends messages to other nodes or to the configuration manager when necessary. The main variables and messages are those that concern the BSet. The configuration manager computes the original BSet (in the same way as the passive approach, since it is the quiescent set) and the so called PSet (short for *primed set*), the set of all nodes that may be recipients of transactions initiated by BSet members. Whenever such a node receives a message from a BSet member, it informs the configuration manager that it must be added to the (extended) BSet. The configuration manager calculates the new PSet and sends the updated value of the BSet to all those nodes.

## 2.2 Discussion

The authors of the approaches just described discuss their results, but since [GK96] contains not a single example, comparison with [KM90] is stated in brief and vague terms. Since we use the same framework, we take a closer look at the two different methods in order to gain a better insight into the reconfiguration process to achieve further reduction in disruption. Both approaches are analysed in terms of disruption, run-time overhead, implementation, and how they deal with hierarchic systems.

### 2.2.1 Implementation

In the first approach the application programmer is expected to provide code that allows the component to reach the passive state and keep it, whereas in the second approach this happens transparently due to the added assumption that nodes are consistent when there are no interactions with the rest of the system. Of course, the other side of the coin is that the implementation is hidden away into the hooks which must trap low level events like message arrival.

On the other hand, the first approach requires just one `passivate` message for each node in the extended passive set, while the second method generates initially $|PSet_0|$ messages and then $1+|PSet_i|$ messages for the addition of the $i$-th member of the extended BSet. Also, the messages in the static method are just tokens, while those in the dynamic approach are descriptions of (potentially large) sets. The blocking approach has thus much greater run-time overhead and is more complex to implement, but imposes less burden on the component programmer.

### 2.2.2 Disruption

The important point in any dynamic reconfiguration method is that the "freezing" of a node N does not prevent other nodes from reaching their "frozen" state. Basically, both approaches solve the problem by "freezing" also every node that depends on N or on which N depends. This does not minimise disruption and in fact may involve many components besides those that are affected by reconfiguration.

To illustrate the differences between both approaches, let us apply them to common examples. We write OBS and EBS for the original and the extended BSet, respectively. The first example is the system of Figure 2.2 on page 11. If the server is to be replaced, we have seen that the first approach passivates all nodes. The second method considers $OBS = \{M, S\}$, $EBS = \{\}$ and $PSet = \{S\}$, because the only outgoing transaction from a BSet member is p and goes to the server. This means that on occurrence of p, $EBS = \{S\}$ and on its completion both M and S block because they are idle and in the BSet. Notice that the clients are not blocked and therefore may initiate new transactions during reconfiguration. Since M is blocked, it will queue the requests and service them after the changes done to the system. This was considered an inconsistency in the passive approach, but in our opinion this is perfectly acceptable because the server manager has not been changed. Therefore its interface with the agents and the new server is the same as previously. This means that the new server is able to attend requests sent to the old one. To sum up, the blocking approach causes less disruption. The reason is that a passive node is active regarding transactions it services. Therefore, to achieve quiescence the node must be "shielded" from outside requests and that shield (the extended passive set) must remain during reconfiguration. No such shield is required in the second approach since the components actually stop.

Now consider the same system but where the first client has to be replaced. In this case $QS = EPS = \{C_1\}$ since no component depends on $C_1$. Therefore, as soon as the

transaction terminates, $C_1$ will become passive and automatically quiescent. Reconfiguration can start, while all other components remain active. Applying the blocking approach one has $OBS = \{C_1\}$, $PSet_0 = \{A_1\}$, $EBS_0 = \{\}$. If there is a pending client request, $EBS_1 = \{A_1\}$ and $PSet_1 = \{M\}$. Since the transaction is dependent, after two more steps $EBS_3 = \{A_1, M, S\}$. In other words, to replace a client, the server is blocked (even if temporarily)! In this example the first approach causes much less disruption, contrary to the claim in [GK96] that the blocking approach performs always at least as well as the passive method. The reason is that the blocking approach is purely dynamic: it does not precompute the dependencies between nodes, which is essential to determine whether the blocking of two components will interfere with each other. Therefore at run-time the method goes through *all* the nodes an OBS member depends on, which form the EBS. If the configuration manager would compute the paths between OBS members, disruption could be greatly reduced in most cases.

The last example is the left half of the system in Figure 2.5 on page 13. In the second approach, if A is not engaged in any transaction with B, it will block immediately. Thus as soon as D is idle it will get blocked too and reconfiguration starts. In the first approach, all nodes from B to C will be passivated even if no dependent transaction will occur. This is the advantage of a dynamic method. It only takes into account transactions that are actually occurring in the running system, while a static analysis must involve all transactions that *may* occur.

The authors have concentrated on the number of nodes that are passivated or blocked by their methods, but we think that indirect disruption must also be taken into account. Since a blocked node does not any processing whatever, any transaction it services or initiates unrelated to the reconfiguration will also be stopped and that may lead to (partial) inactivation of other components. Since passive nodes still service requests they cause indirect disruption in smaller scale. But internal processing that requires initiation of transactions is still hindered. This is recognised in [KM90]. The authors observe that the replacement of the server in Figure 2.2 on page 11 passivates the clients thus stopping them from interacting with other nodes not shown on the figure. Therefore, they should only be passive with respect to the server being replaced, not other nodes unrelated to the change. This could be achieved by distinguishing the relevant connections and modeling their state (connected-passive, connected-active, disconnected). This would allow more granularity, but the authors argue it would lead to more complex substates and more complex actions to obtain consistency since the nodes would be partially active. Therefore they conclude that their approach, while not minimal, is simple and sufficient.

In our opinion there is another factor that contributes to a greater disruption than necessary in some cases: the requirement for quiescence of the initiator node in (un)link changes.

Let us assume that the change specification contains a command `unlink` N `from` N′ for a non-consequent transaction. It is not necessary for N to be quiescent. It is enough to be passive, thus not starting any new transaction with N′. Consider the right subsystem of Figure 2.5 on page 13 where connection 1 will be removed. If $F \in QS$ then E and every node that depends on transaction 2 would be in EPS. Therefore they and all nodes on which they can initiate transactions would be partially inactive. If F were only passivated, the extended passive set would not include the other nodes, reducing direct and indirect disruption greatly.

Also, if there is a `link` N `to` N′ command but neither N nor N′ are changed, then the new connection is the replacement of a previously existing connection or it is an optional connection (because N was already working without it). In any case N does not have to be quiescent or even passive. It is only in those states if it has to be replaced or if some of its connections will be removed. In our opinion, the addition of connections by itself should not impose passiveness.

Both approaches measure disruption only in terms of nodes, neglecting the time factor. In the configuration model described in Section 2.1 on page 9, first components are "frozen", then change commands are applied, and finally components are activated. This does not minimise disruption time because each phase can only begin after the previous one ended. Moreover, commands are performed in a fixed sequence (first all `unlink`, then all `remove`, etc.). It is obvious that in many cases some changes are independent of others. In those cases a part of the system might be changed without having to wait for nodes in other parts to be "frozen", or commands of different kinds might be performed in parallel.

### 2.2.3 Hierarchic Systems

In [GK96] no reference is made to hierarchic systems. In fact, the blocking approach does not work for them since a composite node will in the general case interleave transactions, because its subcomponents run in parallel. As written before, [KM90] deals with such systems but their treatment is still very sketchy. Basically, it only indicates how to compute a composite node's dependencies from its subcomponents. From there the extended passive set at the higher level can be computed. If the composite node has to be passivated, all its subcomponents must also. This certainly does not minimise disruption. We also feel that requiring independent transactions between composite components (as in CONIC [MKS89]) to reduce the number of those to be passivated may lead to extremely large components or to many small ones. In any case it may force the system designer to partition the system into artificial composite components that are uneasy to work with. But more importantly, [KM90] does not deal with the interaction between changes at different levels or how changes at a lower level will affect higher levels of the component hierarchy.

## 2.3 The Refined Model

Although the original analysis of the requirements for dynamic configuration [KM85] stressed the importance of modularity and well-defined component interfaces, the model presented in [KM90] does not provide any details about it. However, the concrete configuration language presented in [KM85], CONIC, and its successor DARWIN [MDEK95] provide a mechanism to specify the communication points of a component, called ports. Our model will thus support that notion, too. An interface is just a set of ports, each being used either to initiate transactions or to receive requests. Since the environment has no access to the inner structure of a component, the programmer must provide in the interface the dependencies between initiator ports and recipient ports.

**Definition 2.1.** A *node interface* is a triple $\langle I, R, D \rangle$ where

- $I$ is the finite set of *initiator ports*;
- $R$ is the finite set of *recipient ports* such that $I \cap R = \emptyset$;
- $D \subseteq R \times I$ is the *port dependency* relation. □

A system is simply a set of connected nodes, where a connection is given by an initiator port and a recipient port. To capture sound software engineering principles (modularity, encapsulation, data hiding, etc.), a system has no access to the inner structure of its nodes; it knows only their interfaces.

The original model is intended for node based reconfiguration, i.e., "freezing" is done upon nodes. Moreover, the computation of the passive and blocked sets depends only on the pattern of connections, not on their number. Therefore the model can assume

without loss of generality that there is only one arc between a given pair of components. A connection based approach like ours distinguishes individual transactions and thus one must allow several connections to be linked to the same port (but only one transaction for any given pair of ports). This covers typical situations like client-server (all client transactions linked to same server recipient port) and broadcast (many transactions with common initiator port). To avoid deadlock, the connections (together with the port dependencies) may form no cycle. Formally, there may be no closed sequence of alternating recipient and initiator ports such that every initiator port is linked to the succeeding recipient port which in turn depends on the next initiator port in the sequence.

**Definition 2.2.** A *system* is a pair $\langle N, T\rangle$ where

- $N$ is a non-empty finite set of node interfaces;
- $T \subseteq \bigcup_{n\in N} I_n \times \bigcup_{n\in N} R_n$ is the set of *transactions*.

A *non-empty path* is a sequence of ports $r_1 i_1 r_2 i_2 \cdots r_m i_m$ with $m > 0$ such that

- $\forall j \in \{1,\ldots,m-1\}\ \langle i_j, r_{j+1}\rangle \in T$;
- $\forall j \in \{1,\ldots,m\}\ \exists n \in N\ \langle r_j, i_j\rangle \in D_n$.

For every non-empty path $r_1 \cdots i_m$ one has $\langle i_m, r_1\rangle \notin T$. □

The original model assumes that the dependencies among transactions are given with the system. We feel that our notion of port dependency is more realistic and more flexible since it allows the system architect to work with components programmed by several people, which may not include himself. Besides, it is a more primitive notion because the dependencies among connections can be computed from those between ports: if recipient port $r$ depends on initiator port $i$, then *any* transaction received by $r$ starts a transaction (i.e., depends) on *every* connection from $i$. As in the original model, the inverse is not true: the component might start a transaction on port $i$ without having received any request on port $r$. The transaction dependency relation is closed under transitivity.

**Definition 2.3.** Given a system $\langle N, T\rangle$, the *transaction dependency* relation $/ \subseteq T \times T$ is defined as $\langle i, r\rangle/\langle i', r'\rangle \Leftrightarrow (\exists n \in N\ \langle r, i'\rangle \in D_n) \vee (\exists t'' \in T\ \langle i, r\rangle/t'' \wedge t''/\langle i', r'\rangle)$.

A transaction $t$ is *dependent* (on the *consequent* transaction $t'$) if $\exists t' \in T\ t/t'$, otherwise $t$ is *independent*. □

The acyclic condition on port paths can thus be restated as: transaction dependency is anti-reflexive.

**Proposition 2.1.** *In a system* $\langle N, T\rangle$, $\nexists t \in T\ t/t$. □

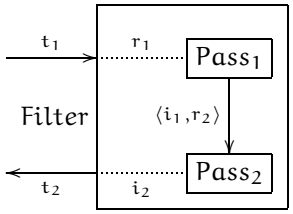
*Proof.* It is easy to see that whenever $t/t'$, with $t = \langle i, r\rangle$ and $t' = \langle i', r'\rangle$, there is a non-empty path from $r$ to $i'$. It is immediate for the base case, and whenever $t/t''$ and $t''/t'$, the non-empty paths $r \cdots i''$ and $r'' \cdots i'$ can be concatenated, forming a new non-empty path. Therefore, if we had $t/t$, the path $r \cdots i$ would be cyclic: $\langle i, r\rangle = t \in T$. ✓

To build modular architectures it must be possible to abstract systems into nodes which will be part of other systems. A system is encapsulated in a composite node by hiding part of the system's ports. The dependencies of the remaining visible ones (i.e., the ports of the composite node) are given by the underlying system.

**Definition 2.4.** A *composite node* consists of an interface $\langle I, R, D\rangle$ and a system $\langle N, T\rangle$ such that

- $I \subseteq \bigcup_{n \in N} I_n$;
- $R \subseteq \bigcup_{n \in N} R_n$;
- $D = \{\langle r, i\rangle \in R \times I \mid r\, i_1 \cdots r_m i \text{ is a non-empty path in } \langle N, T\rangle\}$. □

*Example 2.1.* Consider the two-pass filter depicted below with $Pass_n = \langle\{i_n\},\{r_n\},\{\langle r_n, i_n\rangle\}\rangle$ for $n = 1, 2$. The dotted lines indicate for each port of a composite node which is the corresponding port of the contained system.




Since $r_1 i_1 r_2 i_2$ is a non-empty path, we have $Filter = \langle\{i_2\},\{r_1\},\{\langle r_1, i_2\rangle\}\rangle$ and hence $t_1/t_2$. This shows how inner port dependencies entail outer transaction dependencies. □

If a node is not decomposed into further nodes, then it is called *simple*. Formally, only its interface is available. A system is *hierarchic* if it contains at least one composite node. Strictly speaking, given a system it is impossible to know for any of its nodes whether it is simple or composite because the formal definition of a system only provides the node interfaces. Thus it is possible for a simple node to be changed into a composite one and vice-versa in a transparent manner to the system.

## 2.4 Minimising Disruption

From the long summary and analysis of the passive and blocking approaches it becomes clear that to minimise disruption we must look for a static blocking method at the connection level. To ensure that blocking a connection will not prevent others from reaching the blocked state, we can use a previously mentioned idea: to order the execution of "freeze" commands. This works at the connection level because transactions do not form cycles. Extending the execution ordering to all commands one can define precisely what changes may be performed in parallel to reduce disruption time.

### 2.4.1 The Connection Approach

The essence of our proposal is to block only those connections that will be removed. To block a connection its initiator node waits for any ongoing transaction (on that connection) to finish and then simply does not start a new one. For this to work we assume, as in the original model, that a transaction finishes in finite time and that its initiator knows when it ends. A simple implementation might be the following. For each component, assign to each transaction $T_i$ it might initiate a boolean variable `blocked[i]` initialised to false and a semaphore `S[i]` to ensure that the transaction completes before being blocked. Then substitute the transaction code $T_i$ by

```
P(S[i]);
wait while blocked[i]; Ti;
V(S[i]);
```

and add the following case to the code that dispatches the incoming requests:

```
if msg.command = block then begin
  i := msg.arg; P(S[i]);
  blocked[i] := true; V(S[i]);
  send(config_manager, blocked, i)
end
```

This code can also be provided by three hooks if wished. One to be called on transaction begin, one on transaction end. These hooks must be explicitly called by the component's programmer, passing the transaction identifier as argument. The third hook would be called transparently to the component on message arrival. Compared to the blocking approach, run-time overhead is small since only one simple `block` message per connection is sent and acknowledged. However, the number of messages is usually larger than in the passive approach because each node to be removed has to receive as many `block` messages as the connections it has.

Blocking a connection means that the node will not service any transaction that depends on the blocked one. To ensure that the blocking of one connection will not prevent other pending transactions to block, the configuration manager orders the blocking according to dependency: if transaction $t$ depends on $t'$ then the `block` message is sent to the initiator of $t'$ only after $t$ is known to be blocking. This is always possible because transactions do not depend cyclically on each other.

*Example 2.2.* Consider again the client-server system of Figure 2.2 on page 11. Let us assume that $C_1$ and the server will be replaced. Then $s_1$ and $p$ must block because they will be removed, but $p$ cannot simply block at once because it may have to service a pending $s_1$ request (or else $s_1$ could never terminate and get blocked). Therefore, blocking $s_1$ before $p$ we are sure that the blocking state is reachable for each link. Also, any request received by manager $M$ after $p$ blocked can be safely queued until the server has been replaced because it is known that any connection that depends on $p$ and that had to block has already done so. □

Notice that this method would not work if the server would be allowed to be simply removed without being replaced by a new one. In that case a partially completed $s_2$ request could remain after reconfiguration: clearly an inconsistent state. We assume that the validation process has ruled out such cases. If a consequent transaction is removed, either a replacement connection is created or else the transactions which depend on the removed one are changed too.

The original reconfiguration model distinguished two kinds of commands: those that are given in the change specification (`create`, etc.) and those that are used to "freeze" the components (`passivate`, `block`). The former are common to the passive and blocking approaches, while the latter are specific to each approach. In our model the "freeze" command blocks a connection. Furthermore, as multiple transactions are allowed between the same pair of components, the syntax of the `(un)link` commands has to change slightly.

**Definition 2.5.** A *command* is either of `create` $n$, `remove` $n$, `link` $t$, `unlink` $t$ or `block` $t$, where $n$ is a node interface and $t$ a transaction. □

### 2.4.2 The Partial Order

To minimise disruption time, the precise execution of the commands issued by the configuration manager is given by a temporal order $<$: if $c < c'$ then command $c'$ can only be executed after command $c$ has completed. Commands that are not related through the ordering can be executed in parallel. It is obvious that the order must include the following relationships:

1. If a transaction $t$ depends on a transaction $t'$, then $t$ must be blocked before $t'$.

2. A connection must be blocked before it is removed.

3. A node can only be removed after its connections have been removed.

4. A node can only be linked after its creation.

We will be conservative and impose a further restriction. In some systems it might not be necessary and thus further parallelisation can be achieved. Consider a simple system with a client linked through transaction $c$ to a server. If the server is to be replaced then a new connection $c'$ is needed. However, since the client remains the same, the communication protocol with the new server is the same as with the old one. Therefore, $c'$ is the same transaction as $c$ and we feel it does not make sense to link $c'$ before unlinking $c$. Besides, it might lead to execution errors if the implementation of the client assumes that there is always only one connection on that particular port. The general rule is:

5. A connection for transactions initiated by a node can be established only if no more connections from it will be removed.

As can be seen by exhaustive inspection of all possible interactions between the existing kinds of commands (`block`, `link`, `unlink`, `remove`, `create`) no further rules are necessary since there are no other dependencies between the commands and thus they may run in parallel.

**Definition 2.6.** Given a set of commands $C$ for a system $\langle N, T\rangle$, the *command order* $< \subseteq C \times C$ is the smallest relation that satisfies

1. `block` $t <$ `block` $t'$ if $t/t'$ and $\nexists$`block` $t'' \in C\ t/t'' \wedge t''/t'$;

2. `block` $t <$ `unlink` $t$;

3. `unlink` $\langle i, r\rangle <$ `remove` $n$ if $i \in I_n$ or $r \in R_n$;

4. `create` $n <$ `link` $\langle i, r\rangle$ if $i \in I_n$ or $r \in R_n$;

5. `unlink` $\langle i, r\rangle <$ `link` $\langle i, r'\rangle$. □

Since the configuration manager directly implements the command order, it is desirable to avoid redundancy. Therefore the ordering is an immediate precedence relation: if $c < c'$ then there is no command $c''$ such that $c < c'' < c'$. Due to the nature of the five cases this could only happen with `block` commands (case 1). Therefore the definition above imposes the additional condition.

*Example 2.3.* Applying the definition to Example 2.2 on the page before, only 4 steps are necessary to replace the first client and the server whereby each step consists of several commands executing in parallel:

1. `create` $C_1'$, `block` $s_1$, `create` $S'$

2. `unlink` $s_1$, `block` $p$

3. `link` $s_1'$ (the connection from $C_1'$ to $A_1$), `remove` $C_1$, `unlink` $p$

4. `link` $p'$ (the connection from $M$ to $S'$), `remove` $S$

Notice that in some cases some commands of step $i+1$ can start without step $i$ being completed (the exact order is given in Example 2.4 on page 22). □

To summarise, a connection based approach is not only advantageous in terms of the number of parts being "frozen", but also in terms of minimising disruption time. In fact, in the node based approaches several nodes have to "freeze" just to let those nodes that really matter for the reconfiguration to become quiescent. In practice this means that reconfiguration can only start after all nodes have "frozen". We think it is possible to have rules that allow one to calculate the exact set of nodes that have to "freeze" for a given change command to be executed, but those rules would be much more complicated than those shown above. Given that in a connection based approach the number of parts to be "frozen" is much smaller, and that "freeze" and change commands can be better interleaved, we conclude that our method can reduce disruption time considerably.

## 2.5 The Configuration Manager

Since a configuration manager executes several commands with some dependencies among them, we observe that such a manager can be seen as a parallel system too, with components and transactions. The goal is to have a precise definition of a configuration manager for a given set of commands to be applied to a given system. In this way the same framework can be used both for managers and the systems they reconfigure. In particular, the definition to be obtained can serve as a basis for a straightforward implementation of configuration managers, although our main goal is to provide a system view of a manager. To facilitate exposition we start with flat systems.

### 2.5.1 Flat Systems

The basic idea is that each change command is implemented by a component, and connections between components make dependencies between the corresponding commands explicit. To be more precise, if $c < c'$ then the component corresponding to $c'$ will initiate a transaction with the component corresponding to $c$. The transaction can be seen as a request from $c'$ to execute $c$. Once the acknowledgment is received, $c'$ can execute. If $c'$ depends on several commands it must wait for all its requests to be attended. A command is executed only once, even if several other commands are connected to (i.e., depend on) it.

A component implementing a command $c$ must therefore have two ports. The recipient port $s_c$ receives all requests from the successors of $c$, i.e., those nodes that can only execute after $c$. The initiator port $p_c$ sends requests to all predecessors of $c$ to start execution and waits for the acknowledgments. It is obvious that $s_c$ depends on $p_c$.

In some cases a command $c$ does not depend on the execution of others. In other words, there is no $c'$ such that $c' < c$. The inverse can also happen: no $c'$ depends on $c$. For example, if connections are to be removed, there is always at least one `block` command to be executed first (i.e., it depends on no other one) and at least one `block` to be executed last. In these cases the corresponding components only need one port. Instead of providing special component definitions we take a generic approach. The configuration manager has always one special `nop` component with one recipient port $s_{\mathsf{nop}}$ and one initiator port $p_{\mathsf{nop}}$ (like any regular component) but there is no dependency between them. For any request received by $s_{\mathsf{nop}}$ an acknowledgment is immediately sent. Likewise, any transaction linked to $p_{\mathsf{nop}}$ is immediately started. To see why this works, consider the case where there is no $c'$ such that $c' < c$. Since $c$ depends on no other command, it can execute at once. In other words, the fact that $c$ has no predecessor can be seen as its predecessor being the "empty" command `nop`. Therefore, if $c$'s predecessor port $p_c$ is linked to the successor port $s_{\mathsf{nop}}$, the request from $c$ is immediately attended by `nop` and therefore $c$ can execute at once as wished.

**Definition 2.7.** The *system configuration manager* that reconfigures the flat system $\langle N, T\rangle$ according to commands $C$ is a system $\langle N', T'\rangle$ where

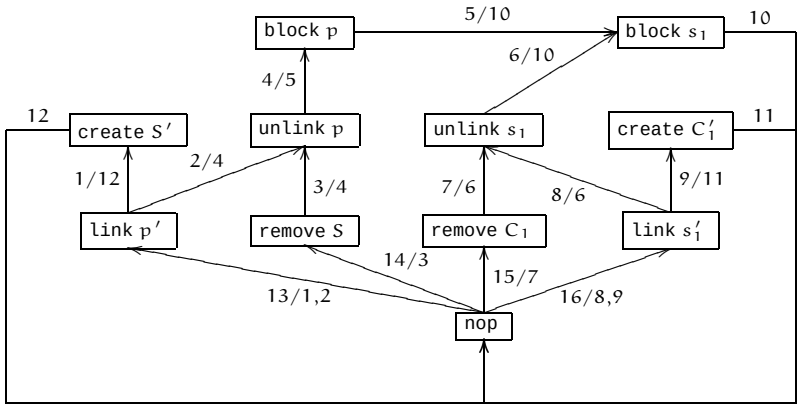
$$\begin{aligned} N' &= \{\langle\{p_c\},\{s_c\},\{\langle s_c, p_c\rangle\}\rangle \mid c \in C\} \\ &\cup \{\langle\{p_{\mathsf{nop}}\},\{s_{\mathsf{nop}}\},\emptyset\rangle\} \end{aligned}$$

and $T'$ is the smallest relation that satisfies

1. $\langle p_c, s_{c'}\rangle \in T'$ if $c' < c$;
2. $\langle p_c, s_{\mathsf{nop}}\rangle \in T'$ if $\nexists c'\ c' < c$;
3. $\langle p_{\mathsf{nop}}, s_c\rangle \in T'$ if $\nexists c'\ c < c'$. □

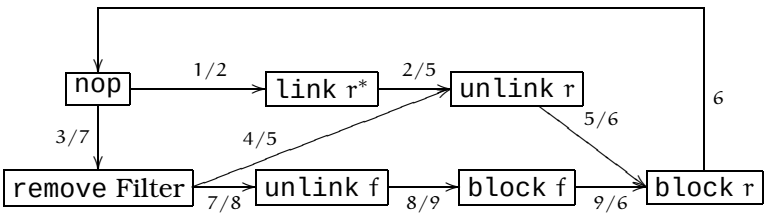
It is possible to simplify matters by omitting $p_{\mathsf{nop}}$ and the third condition, since a command $c$ without successors does not get any request in its $s_c$ port from any other command and therefore the port may be left unlinked. However, we chose the above presentation to emphasize the uniformity of the approach.

*Example 2.4.* According to this definition the reconfiguration of Example 2.2 on page 19 can be done by the following manager which allows us to quickly see the ordering of the commands, in particular which must be executed sequentially and which can run in parallel.




Notice that the execution path starts and ends at nop. The arrows show the direction of requests, it is upon acknowledgments (in the opposite direction) that commands are executed (compare with the partial order in Example 2.3 on page 20). Hence, a sequential (ie, total) execution order for the commands can be obtained by removing the nop node and its arcs, reversing the direction of the other arcs, and then computing a topological sort [CLR90]. □

*Example 2.5.* Consider the system Client $\xrightarrow{r/f}$ Filter $\xrightarrow{f}$ Server. To remove the filter and link the client directly to the server with connection $r^*$ we execute the following commands:



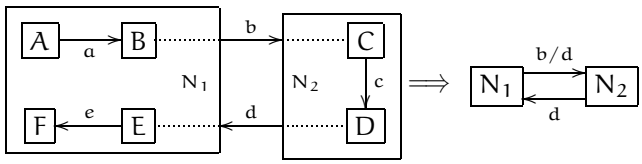

Note that link $r^*$ > unlink $r$ because those transactions have the same initiator. □

The graphical representation of the configuration manager makes it easy to have an estimate of the upper bound of the time needed for reconfiguration. First assign to each transaction a numeric weight representing the upper bound for executing the command corresponding to the initiator of the given transaction. The bound may only take the command type into account (`link`, `create`, etc.) or it may be specific to the component or connection operated upon. Transactions initiated by `nop` have weight zero. The time needed for reconfiguration is then the longest path from `nop` back to `nop` [CLR90].

### 2.5.2 Hierarchic Systems

Hierarchic systems pose a problem that does not occur in flat systems: if commands $c$ and $c'$ apply to different subsystems, it might still be the case that $c < c'$ (or vice-versa) due to the way the subsystems are connected.

*Example 2.6.* Consider the hierarchic system presented below. Assume that for each of $B, C, D$, and $E$, the recipient port depends on the initiator port. The same applies to $N_2$, according to the definitions of composite nodes and transaction dependency, and thus $b/d$ as seen on the right. In other words, $N_2$ is like the two-pass filter of Example 2.1 on page 18.




For this configuration one has $a/e$, which is not apparent just by looking at $N_1$. If $A$ and $F$ are to be replaced, those two connections cannot be blocked in parallel. One could flatten the whole system to discover that `block` $a <$ `block` $e$, but that defeats the whole purpose of building a modular system. □

To reflect the hierarchy of a system and the benefits of its partitioning, we propose a configuration manager for each node. We have seen that a manager for a system is a system itself. Likewise, the manager of a node will be a node, with an interface that will allow it to be linked to other configuration managers. The problem is thus what interface a node manager needs and how should it be linked to other managers. The goal is to achieve the correct order of command execution in the most modular possible way. In other words, a node manager should only know about the subsystem it manages, not about the managers it is linked to. Therefore, the internal structure of a node manager should be such that it can work in any possible context.

The solution to the problem is based on the following observations. Let us assume that node $x$ has an initiator port $i$ linked to recipient port $r$ of node $y$. Thus any change inside $x$ that depends on transaction $\langle i, r\rangle$ must occur before the changes inside $y$ that depend on $\langle i, r\rangle$. Therefore the requests of the change commands inside $y$ must be acknowledged by the change commands inside $x$. Therefore the direction of requests is opposite to the direction of the transaction that establishes the dependency between $x$ and $y$. To sum up, the configuration manager $x'$ for $x$ has *recipient* port $i$, the manager $y'$ for $y$ has *initiator* port $r$, and the requests of $y'$ are passed to $x'$ through transaction $\langle r, i\rangle$. If the system's transaction $\langle i, r\rangle$ is going to be blocked then the reconfiguration manager for the whole system cannot just link sub-manager $x'$ to sub-manager $y'$. In this case port $r$ of $y'$ is connected to the recipient port of `block` $\langle i, r\rangle$ whose initiator port is linked to port $i$ of $x'$.

To put it in more general terms, the configuration manager for a hierarchical system $S$ consists of one component for each command, one component called `nop`, and one configuration manager for each node in $S$. A node manager has the same interface as the node whose reconfiguration it manages, except that initiator ports are exchanged with recipient ports. This implies that the manager's port dependency relation is the

inverse of the node's dependencies, and that connections among node managers are the opposite of those between nodes, except for transactions that must be removed. Those will be of course substituted by the respective `block` command.

**Definition 2.8.** The *system configuration manager* that reconfigures the hierarchic system $\langle N, T\rangle$ according to commands $C$ is a system $\langle N', T'\rangle$ where

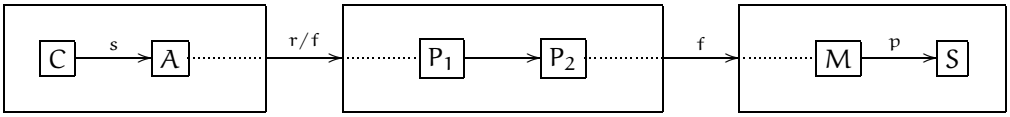
$$\begin{aligned} N' &= \{\langle R_n, I_n, D_n^{-1}\rangle \mid n \in N\} \\ &\cup \{\langle\{p_c\},\{s_c\},\{\langle s_c, p_c\rangle\}\rangle \mid c \in C\} \\ &\cup \{\langle\{p_{\mathsf{nop}}\},\{s_{\mathsf{nop}}\},\emptyset\rangle\} \end{aligned}$$

and $T'$ is the smallest relation that satisfies

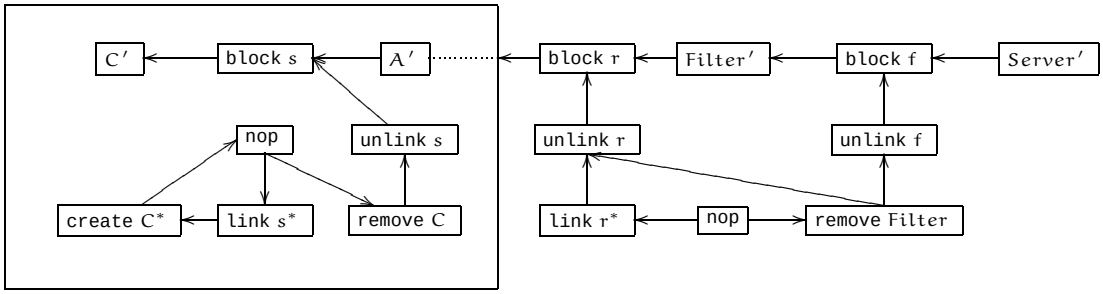
1. $\langle r, i\rangle \in T'$ if $\langle i, r\rangle \in T \wedge \texttt{block}\ \langle i, r\rangle \notin C$;
2. $\langle r, s_c\rangle, \langle p_c, i\rangle \in T'$ if $c = \texttt{block}\ \langle i, r\rangle$;
3. $\langle p_c, s_{c'}\rangle \in T'$ if $c' < c$;
4. $\langle p_c, s_{\mathsf{nop}}\rangle \in T'$ if $\nexists c'\ c' < c$;
5. $\langle p_{\mathsf{nop}}, s_c\rangle \in T'$ if $\nexists c'\ c < c'$. □

**Definition 2.9.** The *node configuration manager* for a composite node with interface $\langle I, R, D\rangle$ and system $\langle N, T\rangle$, to which commands $C$ will apply, is a composite node with interface $\langle R, I, D^{-1}\rangle$ and system configuration manager $\langle N', T'\rangle$. □

*Example 2.7.* Let us assume that the components in Example 2.5 on page 22 are composite, with the two-pass filter of Example 2.1 on page 18, the client and its agent (see Figure 2.2 on page 11) being put together, and similarly for the server and its manager.




Assume further that in addition to the reconfiguration of Example 2.5 on page 22 we replace the client C by C*. The resulting manager is




□

For the definition to be complete it remains to be said how a configuration manager $x'$ for a simple node $x$ behaves. As for any node manager its interface is the "mirror" of the node's interface, and the same happens to port dependencies. If recipient port $r$ of the manager $x'$ depends on initiator port $i$, then $x'$ must forward any request received on $r$ to port $i$[2]. If $r$ does not depend on any initiator port, then $x'$ acknowledges immediately any request received by $r$. As usual, all transactions connected to a initiator port are also immediately started.

[2] A composite node manager also does this, the only difference being that the forwarding is done through the dependency path made explicit by the composite node's architecture.

*Example 2.8.* Returning to Example 2.6 on page 23, let us assume that all nodes from A to F are simple. Then a can be blocked at once since A′ attends the request made by `block` a. F′ issues a request to `block` e which gets forwarded by C′ and D′ until A′ which gets immediately acknowledged at that point. In a slightly optimised implementation of this model, if `block` a had already executed, it would acknowledge `block` e's request without forwarding it to A′. As a further example, consider that there is no dependency between the ports of C (i.e., b is independent of d)., Then `block` e is acknowledged at the recipient port of C′ and therefore a and e can be blocked in parallel as desired. □

## 2.6 Concluding Remarks

We have adopted a simple and general framework at the software architecture level stating which parts of the system should be "frozen" in order to achieve a stable consistent state and how the "freezing" and the changes are performed. We analysed, formalized, refined, and extended the framework in order to minimise disruption and to handle hierarchic systems.

In fact, switching from a component based to a connection based approach, we have come up with a minimal solution (since it only blocks the connections that will be removed) that is conceptually very simple. On the other hand, for the first time for this framework, we have concentrated on the time taken by the reconfiguration process. In particular we have defined an order for the change commands that may reduce, considerably, the disruption of independent parts of the system being reconfigured. The assumption is, again, that commands may be executed in parallel.

Since a configuration manager executes the commands of a given change specification, it can be seen itself as a system of interconnected components, where a component is a single change command and a connection denotes the dependency between the two commands it links together. This model gives a precise and complete account on how a configuration manager may execute a change specification. The model is also particularly useful for hierarchic systems, showing how the reconfiguration process of the whole system can be obtained simply by connecting the configuration managers of the subsystems together, in a way that mirrors the connections between the subsystems.

The work presented in this chapter was done in early 1997. Since then the blocking approach described in [GK96] has been made part of a complete framework for dynamic reconfiguration [MG99]. Its main characteristic is the use of an object-oriented reconfiguration language, in this case Java with pre-defined methods for querying and changing the current configuration. For each component type of the application (e.g., client, server, and filter, in our examples) there is a class to be used by the reconfiguration manager. The class contains the information needed by the manager to reconfigure systems containing components of that type. For example, the class contains one variable for each port of the component type. For composite components, the class also contains one method for each possible reconfiguration. Each method uses the primitive commands (like `unlink`) or the methods of the contained components. Each method corresponds thus to a node configuration manager in our model. In fact, it corresponds to many managers: the method can adapt the actual reconfiguration to be done to the current configuration of the composite node, because the method is written in a concrete programming language and therefore may use query primitives and control structures (like while-do, if-then, etc.). For example, a method can replace all sub-components of a filter, no matter of how many passes it is made of, while in our abstract model there must be a different node configuration manager for each case. Moreover, an object oriented reconfiguration language allows common reconfigurations to be assigned to a common superclass. For example, it is possible to have an abstract class `Ring` with a reconfiguration method `add` which allows the addition of a new node to a ring topology. Each node would have to belong to a subclass of `Ring`. However, the funda-

mental point to remember is that the idea to mirror the structure of the application at the configuration management level has been validated in practice.

Our uniform view of a configuration manager as a system like the one it manages paves the way for meta-reconfiguration, i.e., a configuration manager might be subject to reconfiguration too. In other words, a change specification can be changed. We think this might be useful in two situations: failures and validation. If an ongoing reconfiguration fails for some reason, then one may try to find another but equivalent (or similar) reconfiguration. Another approach is to undo the changes done so far and re-establish the existing system. In both cases it means that the existing change specification has to be changed while it is being applied, i.e., the configuration manager which is executing the changes must be reconfigured dynamically.

The other situation concerns the validation process. A change in a subsystem (like the removal without replacement of a server) may force some changes to be done in other parts of the system (like substituting a dependent transaction by an independent one). Thus the validation of the change specification for a subsystem may force the change specifications of other subsystems to be changed. On the other hand, this means that the validation of those systems must be redone which may cause further changes in the specifications of other subsystems and so on. Again, change specifications may change dynamically (in this case as they are validated).

# Chapter 3

# The CHAM Approach

The Chemical Abstract Machine (CHAM) [BB92] is a simple, operational, and general-purpose model based on the chemical metaphor introduced by the Gamma formalism [BM96]. The basic idea is to represent the data as a multiset of molecules and the program as a set of multiset rewriting rules that state how the molecules react with each other. A molecule is a user-defined term. There is no control mechanism for the application of rules. The CHAM was first used to specify and analyse the computational behaviour of systems at the software architecture level by Inverardi and Wolf [IW95]. The CHAM was also used to perform architecture refinement [IY96] and deadlock detection [IWY97]. These works describe and study only static architectures: the number and type of components and connections do not change.

Our contribution is to show how the CHAM can be used as a uniform framework to specify, besides computations, architectural styles (following the approach of Le Métayer [Mét98]), different kinds of reconfiguration, and even a restricted form of code mobility. The flexibility provided by molecules allows one to choose the most adequate representation for the architecture and reconfiguration at hand. But we also present fixed syntactic constructions for a CHAM-based ADL and a methodology to specify architectures in a modular way.

The material in this chapter is based on [Wer98b, Wer98c, Wer98a, Wer99].

## 3.1 The CHAM formalism

The chemical model views computation as a sequence of reactions between data elements, called *molecules*. The structure of molecules is defined by the designer. The system state is described by a finite multiset of molecules, the *solution*, written $m_1, \ldots, m_i$. The possible reactions are given by rules of the form

$$m_1, \ldots, m_i \to m'_1, \ldots, m'_j$$

If the current solution contains the molecules given on the left-hand side of a rule, that rule may be applied, replacing those molecules by the ones on the right-hand side. Usually a CHAM is presented using *rule schemata*, the actual rules being instances of those schemata. There is no explicit control mechanism. If several rules may be applied, the CHAM chooses one of them non-deterministically. Reactions on disjoint multisets may occur simultaneously, i.e., in parallel. The solution thus evolves by rewriting steps. A solution is *inert* when no reaction rule can be applied.

As a very simple example, consider a CHAM to build ring-shaped architectures. The initial solution is a multiset of components, each of the form '—c—' stating that it may be connected to two other components. A molecule of the form '—c—c—. . . —c—' may

still grow. When the first component of a linear architecture is linked to the last one, a ring is obtained: 'c—c—. . . —c'. The molecules are thus built as given by the grammar

$$\begin{array}{ll} \textit{Ring} & := \text{`c'}\ \text{`—'}\ \textit{Ring} \mid \text{`c'} \\ \textit{Molecule} & := \text{`—'}\ \textit{Ring}\ \text{`—'} \end{array}$$

and rings are formed according to the reaction rules

$$\begin{array}{c} \text{—x—, —y—} \rightarrow \text{—x—y—} \\ \text{—x—} \rightarrow \text{x} \end{array}$$

If the initial solution has four components, up to four rings can be built. For example, the following transformations lead to two rings.

$$\begin{array}{l} \text{—c—, —c—, —c—, —c—} \longrightarrow \text{—c—c—, —c—, —c—} \\ \qquad\qquad\qquad\qquad\quad \longrightarrow \text{—c—c—, c, —c—} \\ \qquad\qquad\qquad\qquad\quad \longrightarrow \text{—c—c—c—, c} \\ \qquad\qquad\qquad\qquad\quad \longrightarrow \text{c—c—c, c} \end{array}$$

Notice that the second transformation (corresponding to an application of the second reaction rule) could occur in parallel with any of the other transformations.

Rules of the form

$$m_1, \ldots, m_i \rightarrow$$

only remove molecules from the current solution.

Any solution can be considered as a single molecule using the *membrane* operator $\{\!|\cdot|\!\}$. A solution within a membrane can thus be a subsolution of another solution or an argument of a molecule operator. For example, if the molecule constructors are the constant `0` and the unary function `s`, then

$$\texttt{s(}\{\!|\texttt{0,s(0)}|\!\}\texttt{),0}$$

is a solution containing two molecules. Membranes encapsulate solutions and thus force reactions to be local. In other words, the solution inside a membrane evolves independently of the solution outside the membrane. For example, if a CHAM included the rule

$$\texttt{0} \rightarrow$$

then the above solution could be transformed into

$$\texttt{s(}\{\!|\texttt{s(0)}|\!\}\texttt{)}$$

after two (possibly simultaneous) rewriting steps.

The *airlock* operator $\triangleleft$ constructs a molecule $m \triangleleft \{\!|S'|\!\}$ from a solution $S = m \uplus S'$, where $\uplus$ is the multiset union operator. In words, it picks a molecule from a solution and puts the rest of that solution within a membrane. The operator is reversible, which means that $S$ can again be obtained from $m \triangleleft \{\!|S'|\!\}$. For example, using twice the airlock operator it is possible to transform the original solution into

$$\texttt{0}\triangleleft\{\!|\texttt{s(}\{\!|\texttt{s(0)}\triangleleft\{\!|\texttt{0}|\!\}\texttt{)}|\!\}$$

Molecules may permeate through membranes if there are explicit rules for that purpose. For example,

$$\{\!|p \triangleleft M|\!\} \rightarrow p,\ M$$

where M is a solution within a membrane, allows any molecule p to leave the membrane it is within.

Formally, the reaction rules determine a *transformation relation* $\longrightarrow$ between solutions according to the following laws, where S and S′ are solutions and C[·] is a *context*, i.e., a molecule with a hole in which to place another molecule, in particular a solution within a membrane.

**Reaction Law** An instance of the right-hand side of a rule can replace the corresponding instance of its left-hand side. If

$$m_1, m_2, \ldots, m_k \to m'_1, m'_2, \ldots, m'_l$$

is a rule and $M_1, \ldots, M_k, M'_1, \ldots, M'_l$ is an instance of $m_1, \ldots, m_k, m'_1, \ldots, m'_l$ then the solution transformation $M_1, M_2, \ldots, M_k \longrightarrow M'_1, M'_2, \ldots, M'_l$ takes place.

**Chemical Law** Reactions can be performed freely within any solution:

$$\frac{S \longrightarrow S'}{S \uplus S'' \longrightarrow S' \uplus S''}$$

**Membrane Law** A subsolution can evolve freely in any context:

$$\frac{S \longrightarrow S'}{C[S] \longrightarrow C[S']}$$

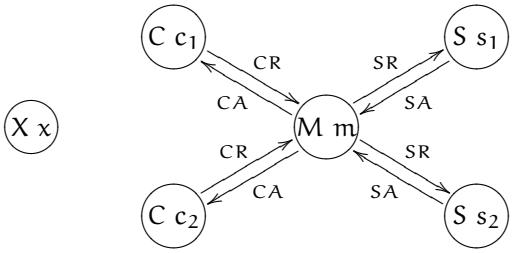
**Airlock Law** A molecule can be extracted and reabsorbed into a solution:

$$m \uplus S \longleftrightarrow m \triangleleft \{\!|S|\!\}$$

## 3.2 The Graph Grammar Approach

We start our exploration with a brief review of Le Métayer's approach to architectural style and evolution specification [Mét98].

Architectures are represented by graphs, the mathematical structure that most closely resembles the intuitive "box and line" drawings. Nodes denote the system components and arcs represent communication links. A graph is a multiset of relation tuples $R(e_1, \ldots, e_n)$ where R is the name of an n-ary relation and $e_i$ are component names. A binary relation $A(e_1, e_2)$ represents a directed arc from $e_1$ to $e_2$ labelled with A. A unary relation $N(e)$ states the role represented by node e in the architecture. The example presented in [Mét98] is the client-server system described by the graph




and, mathematically, by the multiset

$$C(c_i), S(s_i), M(m), X(x), CR(c_i, m), CA(m, c_i), SR(m, s_i), SA(s_i, m)$$

with $i = 1, 2$. Each client (C) sends requests (CR) to a manager (M) that forwards them (SR) to a server (S). The server's answer (SA) gets back to the client (CA). New clients may be created by an external entity (X).

This is just a particular instance of a general class of client-server architectures with exactly one external entity, one manager, and zero or more clients or servers obeying the above connection pattern. Such an architectural style can be specified by a context-free graph grammar. It is, as any context-free grammar, a four-tuple $\langle N, T, P, O\rangle$ where $N$ is a set of non-terminal symbols with a distinguished element $O$ being the origin of the derivation, $T$ is a set of terminal symbols, and $P$ is a set of production rules whose left-hand sides are non-terminals. However, the right-hand sides of the production rules are not sequences, but multisets of (non-)terminals. A (non-)terminal is a relation tuple. The style defined by the grammar is the class of all multisets (i.e., graphs) of terminals generated by the grammar.

The client-server architecture style is defined by

$$\langle\{CS, CS_1\}, \{M, X, C, S, CR, CA, SR, SA\}, P, CS\rangle$$

where $P$ are the rules

$$\begin{aligned} CS &\Rightarrow CS_1(m) \\ CS_1(m) &\Rightarrow CR(c,m), CA(m,c), C(c), CS_1(m) \\ CS_1(m) &\Rightarrow SR(m,s), SA(s,m), S(s), CS_1(m) \\ CS_1(m) &\Rightarrow M(m), X(x) \end{aligned}$$

As expected, the graph presented above can be obtained by this grammar.

The evolution of an architecture is defined by conditional graph rewriting rules. Since a graph is a multiset, those rules are like the guarded multiset rewriting rules of Gamma [BM96]. In this particular case of system evolution, the guard of a rule is a proposition upon the state of the components involved. For the example at hand, [Mét98] assumes that external entities have a boolean variable $newc$ which is set to true when a client is to be created. Dually, a client sets its boolean variable $leave$ when it wishes to leave the system. Client creation and removal can then be described by the rules

$$\begin{aligned} X(x), M(m), x.newc = true &\rightarrow X(x'), M(m), CR(c,m), CA(m,c), C(c) \\ CR(c,m), CA(m,c), C(c), c.leave = true &\rightarrow \end{aligned}$$

The second rule removes a client and its links, while the first one creates a client and links it to the existing manager. Notice that it also replaces the original external entity by a new one. In this way the client creation process (in other words, the internal computation of an external entity) starts over again, thus with $newc$ set to false. If $x$ were not removed, the first rule could be immediately reapplied, possibly leading to an infinite behaviour.

It is obvious that any graph grammar $\langle N, T, P, O\rangle$ is a CHAM, where $N$ and $T$ are the molecules, $P$ the reaction rules, and $O$ the initial solution. Furthermore, deriving a graph is a special case of obtaining a inert solution: since the graphs that represent architectures only contain terminals, no rule can be applied. As it is immediate that conditional graph rewriting rules are basically reaction rules given appropriate molecule syntax to represent the conditions (see, e.g., [IW95]), it follows that the CHAM formalism can be used both for the specification of architecture styles and their evolution. Thus the approach taken in [Mét98], in particular the previous six rules for client-server architectures, can be adopted directly with just a few syntactic changes, without need to use two different, albeit very similar, formalisms.

## 3.3 Ad-hoc Reconfiguration

In the approach just described, the external world is modeled within the system itself. The entities that interact with the system are represented by one or more components

and the events that lead to changes in the architecture are simulated by computations of those components. Put differently, the evolution of the (external) structure of the architecture is based on the evolution of the (internal) state of some components. This means two things. First, only programmed reconfigurations can be simulated. Second, a complete specification of the system's evolution must include a description of each component's data and program, and provide the semantics of the interactions between component and coordinator actions, as done in [Mét98].

We follow instead the Configuration Programming approach, separating structural from computational aspects and using explicit reconfiguration commands to be added unpredictably by the user to the solution. Since the reaction rules will state how components are linked, the user only has to provide *create component* (cc) and *remove component* (rc) commands. Architectural style and reconfiguration are specified by two different CHAMs, the *creation* CHAM and the *evolution* CHAM.

### 3.3.1 Specification

We introduce in the example of the previous section some simplifications in order to omit details which, while making it more "realistic", are not relevant to show how architecture reconfiguration may be specified using the CHAM model. As in the previous chapter, links represent whole *transactions* (i.e., sequences of one or more message exchanges) between components and we assume there is at most one kind of transaction, and thus one link, between any pair of components. This way arcs do not have to be labelled. Another change introduced is merely a matter of taste. Since the component roles of the previous approach correspond to the *component types* of the Configuration Programming approach, we adopt the usual typing notation $c : T$ instead of the relational notation $T(c)$.

The structure of molecules is given by the following grammar, which leaves the precise syntax of component identifiers open.

| | |
|---|---|
| *Molecule* | := *Component* \| *Link* \| *Command* |
| *Component* | := *Id* ':' *Type* |
| *Type* | := 'C' \| 'M' \| 'S' |
| *Link* | := *Id* '—' *Id* |
| *Command* | := 'cc(' *Component* ')' \| 'rc(' *Id* ')' |

To make the example more interesting we assume that there must always be at least one server. The CHAM that specifies the client-server architectural style is

$$\begin{aligned} \text{cc}(m{:}\text{M}) &\rightarrow c{:}\text{C},\ c\text{—}m,\ \text{cc}(m{:}\text{M}) \\ \text{cc}(m{:}\text{M}) &\rightarrow s{:}\text{S},\ m\text{—}s,\ \text{cc}(m{:}\text{M}) \\ s{:}\text{S},\ \text{cc}(m{:}\text{M}) &\rightarrow s{:}\text{S},\ m{:}\text{M} \end{aligned}$$

There are three differences, compared to the original style specification. First, we assume that the manager's name is given by the user, not generated by the system, and thus the symbol CS of the original graph grammar is not necessary. Second, the nonterminal $CS_1$ that keeps track of the manager's name—so that clients and servers can be correctly linked to it—is substituted by a (creation) command. With the elimination of CS, $CS_1$ becomes the origin of the derivation and thus the corresponding cc() command forms the initial solution. Third, the last rule ensures that the architecture creation process only stops when at least one server has been created.

Although syntactically there is no difference between this creation CHAM and the evolution CHAM to be presented next, the distinction can be made just by looking at the rules. A creation CHAM is used to generate *all* architectures belonging to a certain style. This entails two properties of creation CHAMs. First, there are no rc() commands, because components and links are only added, not removed. Second, components may

be created on the right-hand side without a cc() command on the left-hand side, in order to allow an arbitrary number of components (and their connections) to be generated.

Now we turn to the evolution specification. Besides adding and deleting clients as in the original example, we also deal with server and manager creation and removal. Each change must be explicitly invoked by an appropriate command, to be handled by (at least) one reaction rule of the evolution CHAM.

$$\begin{aligned}
cc(c{:}C),\ m{:}M &\rightarrow c{:}C,\ c{—}m,\ m{:}M \\
cc(s{:}S),\ m{:}M &\rightarrow s{:}S,\ m{—}s,\ m{:}M \\
rc(c),\ c{:}C,\ c{—}m &\rightarrow \\
s'{:}S,\ rc(s),\ s{:}S,\ m{—}s &\rightarrow s'{:}S \\
m{:}M,\ rc(m),\ cc(m'{:}M) &\rightarrow m'{:}M \\
m{—}s,\ m'{:}M &\rightarrow m'{—}s,\ m'{:}M \\
c{—}m,\ m'{:}M &\rightarrow c{—}m',\ m'{:}M
\end{aligned}$$

The first four rules deal with client and server creation and removal, while the other rules handle manager substitution, which is indicated by a pair of creation/removal commands. The last two rules relink the existing clients and servers to the new manager. Notice that in this example we assume different variables to be instantiated with different identifiers. Otherwise the right-hand sides would be instances of the left-hand sides. In other words, those two rules could be immediately reapplied (although provoking no change in the architecture) and the solution would never become inert.

This CHAM illustrates ad-hoc reconfiguration because the reconfiguration commands appear only on the left-hand sides of rules. In other words, the commands are only consumed by the CHAM and thus must have been put into the solution by the user. As an example of reconfiguration, let us assume that we have an architecture with a single server (and manager, of course), and we want to add a client and replace the manager. The initial solution for the evolution CHAM is

$$m1{:}M,\ m1{—}s1,\ s1{:}S,\ cc(c1{:}C),\ rc(m1),\ cc(m2{:}M)$$

and the states of the solution until it becomes inert are

1. $m1{:}M,\ m1{—}s1,\ s1{:}S,\ c1{:}C,\ c1{—}m1,\ rc(m1),\ cc(m2{:}M)$
2. $m2{:}M,\ m1{—}s1,\ s1{:}S,\ c1{:}C,\ c1{—}m1$
3. $m2{:}M,\ m2{—}s1,\ s1{:}S,\ c1{:}C,\ c1{—}m1$
4. $m2{:}M,\ m2{—}s1,\ s1{:}S,\ c1{:}C,\ c1{—}m2$

### 3.3.2 Analysis

In general, to make sure that the specification is correct, it is necessary to prove that a creation CHAM can terminate, i.e., that an inert solution can be reached, and that an evolution CHAM does terminate for finite reconfigurations. Usually this involves some assumptions on the initial solution. For our example style we are assuming the initial solution contains just one molecule of the form $cc(m{:}M)$. Then it is quite easy to prove that the creation CHAM may terminate: the third rule consumes the cc() command that is necessary for any of the rules to be triggered.

Another issue is to prove that the architectures generated by the creation CHAM are really those that we intended. Towards that end it is necessary to write down the properties of the architectural style and then, given the initial solution, prove that any inert solution obeys those properties.

Returning to our example, the properties of the client-server style are:

- there is exactly one manager;

- there are $x \geq 0$ clients, each one linked to the manager;
- there are $y > 0$ servers, each one linked to the manager.

We just prove the first and third proposition; the second is similar to the third. As for the first, the rules to create clients and servers maintain the number of cc() molecules in the solution, and the last rule substitutes each one by a manager. Since there is a single molecule in the initial solution, there is a single manager in each architecture belonging to this style. The proof for the third statement is as follows. If the solution is inert, then there is no cc() command because otherwise the first two reaction rules could be applied. Since there is such a command in the initial solution, it must have been consumed somehow. By inspection of the rules, this is only possible by the third reaction rule. However, that rule can only have been applied if there existed a server. Since no rule decreases the number of servers, it is proven that at least one server must exist. As for the server links, the only rule that creates servers connects them to the component whose name is given by the cc() command which is the manager, as observed in the proof of the first property.

Sometimes it is necessary to prove that a reconfiguration does not "break" the style. For some properties this can be done inductively: prove that the initial solution of the evolution CHAM satisfies the property and that each rule keeps it. The first part is usually not needed since it is assumed that the initial solution is either an inert solution of the creation CHAM (and thus satisfies the properties as proven before) or it is the inert solution of a previous reconfiguration (and therefore satisfies the properties as it will be proven by inspection of the rules). It thus suffices to prove that for each rule $L \rightarrow R$, if it is applied to a solution $S$ that satisfies the property, then $S - L \uplus R$ also satisfies it.

As an illustration we prove that the client-server reconfiguration CHAM keeps at least one server. Let $y$ (resp. $y'$) be the number of servers immediately before (resp. after) the application of a rule. One has to prove that $y > 0 \Rightarrow y' > 0$ for each rule. The second rule states that $y' = y + 1$, the fourth rule that $y \geq 2 \Rightarrow y' = y - 1$, and for the remaining rules $y' = y$. It is obvious that for each one the implication is true.

However, the second part of the third property, namely that each server is linked to the manager, cannot be proven in this way because it is not an invariant of the system. In fact, due to rule 5 of the evolution CHAM, the solution does not represent a graph temporarily: there are links $m$—$s$ but there is no $m$! The connectivity property can thus only be established for inert solutions. The proof goes as follows. First show that there is always exactly one manager. Next prove that there is always exactly one connection $m$—$s$ for each server $s$. Finally show that for inert solutions, if $m$:M is the manager and $m'$—$s$ is a server connection, then $m = m'$. The first two statements can be proven inductively, the third results from the fact that in an inert solution the last two rules of the evolution CHAM cannot be applied.

### 3.3.3 Dynamic Reconfiguration

Since a CHAM does not have any control mechanism, the exact order in which the reactions take place is unknown and cannot be predicted. This is no problem if the reconfiguration takes place when the system is shutdown. However, in dynamic reconfiguration the changes occur while the system is running. In that case it is of paramount importance to execute the reconfiguration actions in such a way that the system is kept consistent and that disruption is minimised, as we have seen in the previous chapter. We thus assume there is a configuration manager as described in Definition 2.7 on page 22 capable of executing the reconfiguration commands given in Definition 2.5 on page 19. Notice that our cc() and rc() commands are high-level `create` and `remove` commands, respectively, that also deal with the links "automatically".

We separate concerns by using the CHAM just to specify what to do, letting the configuration manager decide how to do it. To that end we let the CHAM "trace" its execution,

creating a "log" of the changes performed. That log corresponds to the change script that a user would input directly to the validation process of Figure 2.1 on page 10. In other words, the CHAM can be seen as a "compiler" of high-level reconfiguration commands into low-level ones to be executed by the "run-time system", i.e., the configuration manager.

The molecule syntax is extended with

*Command* := '`create` ' *Component* | '`remove` ' *Id* | '`link` ' *Id* '—' *Id* | '`unlink` ' *Id* '—' *Id*

and the evolution CHAM on page 32 becomes

$$\begin{aligned}
\text{cc(c:C)},\ \text{m:M} &\to \text{c:C},\ \text{c—m},\ \text{m:M},\ \texttt{create}\ \text{c:C},\ \texttt{link}\ \text{c—m}\\
\text{cc(s:S)},\ \text{m:M} &\to \text{s:S},\ \text{m—s},\ \text{m:M},\ \texttt{create}\ \text{s:S},\ \texttt{link}\ \text{m—s}\\
\text{rc(c)},\ \text{c:C},\ \text{c—m} &\to \texttt{remove}\ \text{c},\ \texttt{unlink}\ \text{c—m}\\
\text{s}'\text{:S},\ \text{rc(s)},\ \text{s:S},\ \text{m—s} &\to \text{s}'\text{:S},\ \texttt{remove}\ \text{s},\ \texttt{unlink}\ \text{m—s}\\
\text{m:M},\ \text{rc(m)},\ \text{cc(m}'\text{:M)} &\to \text{m}'\text{:M},\ \texttt{remove}\ \text{m},\ \texttt{create}\ \text{m}'\text{:M}\\
\text{m—s},\ \text{m}'\text{:M} &\to \text{m}'\text{—s},\ \text{m}'\text{:M},\ \texttt{unlink}\ \text{m—s},\ \texttt{link}\ \text{m}'\text{—s}\\
\text{c—m},\ \text{m}'\text{:M} &\to \text{c—m}',\ \text{m}'\text{:M},\ \texttt{unlink}\ \text{c—m},\ \texttt{link}\ \text{c—m}'
\end{aligned}$$

The commands have been added in a systematic way. For example, `create` m is added whenever m is a component molecule appearing on the right-hand side but not on the left-hand side of the original rule.

For the reconfiguration steps shown on page 32, the generated change commands are:

1. `create` c1:C, `link` c1—m1,
2. `remove` m1, `create` m2:M,
3. `unlink` m1—s1, `link` m2—s1
4. `unlink` c1—m1, `link` c1—m2,

As mentioned before, if the commands were executed in this order then temporarily some arcs in the system graph would not point to any existing component. Moreover, two commands cancel out: `link` c1—m1 and `unlink` c1—m1. Looking at such a script, it is easy for the validation process to remove opposite commands, and to add the necessary blocking commands. The configuration manager then reorders the commands according to Definition 2.6 on page 20. In this case, one possible sequence is

`block` m1—s1, `create` c1:C, `unlink` m1—s1, `remove` m1,
`create` m2:M, `link` c1—m2, `link` m2—s1

## 3.4 Self-Organisation

The chemical model is well suited to describe self-organising architectures, where external explicit management is kept to a minimum [MK96b]. In fact, the very essence of the CHAM model is that molecules react freely with each other until the solution "stabilises", i.e., becomes inert. Once that state is reached, new reactions may be triggered by adding new molecules, but the reaction process itself is purely an "internal affair". The evolution of the solution proceeds without any intervention from the outside.

Our client-server example illustrates this. Once the commands to substitute the manager are given, the architecture reorganises itself to maintain the right connections, without needing any further commands from the user. In this section we provide a more elaborate example of self-organisation: a n-ary tree architecture where components can

be removed without destroying the properties of the tree. Such a topology might be useful for divide-and-conquer problems, each component splitting the data it gets from its parent and combining the results produced by its children.

The example is a generalisation of the binary tree architecture presented in [YM92]. Following the Configuration Programming approach, [YM92] only considers structural properties. In this case, the *structural integrity* to be kept by the reconfiguration action is the binary tree shape. Such integrity constraints are divided into *node integrity* and *system integrity* properties. The former are local, the latter global. In this example, being a tree is a system constraint because no single node can ensure that the graph is acyclic. On the other hand, it is enough for each node to restrict the number of children to at most two in order to have a binary tree. Other examples of system integrity constraints are the number of components in the tree and the number of trees.

The approach taken in [YM92] to handle structural integrity properties is verification. The properties are expressed as Prolog clauses used to check the architecture after each change. If the reconfiguration violates at least one of the constraints, it must be undone. We follow the self-organisation approach of [MK96b]: when a change occurs, the system components reorganise themselves in order to satisfy the structural constraints.

But first let us specify the $n$-ary tree architectural style. The reaction rules generate trees with a maximum branching factor given by the initial solution. A node is represented by a molecule $n_c^a$ where $n$ is the node's name, $a$ is the number of its ancestors (i.e., its depth), and $c$ is the number of children the node has. A root node has no ancestors and therefore its depth is zero. Natural numbers are represented as usual, using the constant zero and the successor function (written as a postfix $+$). Numbers are compared using substring prefix matching: $n \geq m$ if $n$ is of the form $mx$, and $n > m$ if $n$ is of the form $m + x$, in both cases $x$ being matched by a sequence of zero or more $+$.

$$\begin{aligned} \mathit{Molecule} &:= \mathit{Node}_{\mathit{Nat}}^{\mathit{Nat}} \mid \mathit{Node}\ \text{`—'}\ \mathit{Node} \mid \mathit{Nat} \\ \mathit{Nat} &:= \mathit{Nat}\ \text{`+'} \mid \text{`0'} \end{aligned}$$

The initial solution contains just the root node and the maximal branching factor: $r_0^0$, $0 + +$. The creation CHAM is

$$\begin{aligned} x_c^a,\ c{+}w &\to x_{c+}^a,\ x\text{—}y,\ y_0^{a+},\ c{+}w \\ b{+} &\to \end{aligned}$$

The first rule creates a new node and attaches it to an existing node that has not yet exceeded the children limit. The second rule allows the solution to become inert by eliminating the branch factor.

The initial solution of the evolution CHAM contains the branching factor again and the current architecture. When a node fails, a command to remove it is added to the solution, and the architecture reconfigures itself according to the rules

$$\begin{aligned} \mathrm{rc}(x),\ x\text{—}y,\ x_{c+}^0,\ y_{c'}^{0+} &\to \mathrm{rc}(x),\ y\text{—}x,\ x_c^{0+},\ y_{c'+}^0 \\ \mathrm{rc}(x),\ y\text{—}x,\ x\text{—}z,\ y_{c'}^a,\ x_{c+}^{a+},\ z_{c''}^{a++} &\to \mathrm{rc}(x),\ y\text{—}x,\ y\text{—}z,\ y_{c'+}^a,\ x_c^{a+},\ z_{c''}^{a+} \\ \mathrm{rc}(x),\ y\text{—}x,\ y_{c+}^a,\ x_0^{a+} &\to y_c^a \\ b,\ x\text{—}y,\ x\text{—}z,\ x_{b+w}^a,\ y_c^{a+} &\to b,\ x\text{—}y,\ y\text{—}z,\ x_{bw}^a,\ y_{c+}^{a+} \\ x\text{—}y,\ x_c^a,\ y_{c'}^a &\to x\text{—}y,\ x_c^a,\ y_{c'}^{a+} \\ x\text{—}y,\ x_c^a,\ y_{c'}^{a++} &\to x\text{—}y,\ x_c^a,\ y_{c'}^{a+} \end{aligned}$$

The first rule handles the case of the root node: it is swapped with one of its children, which thus becomes the new root. The former root node now is a middle node or a leaf node and the second or third rule applies, respectively. The second rule links the children of the middle node directly to its parent. The node hence becomes a leaf

node and we get to the third rule which effectively removes the node from the tree, updating the children counter of the parent. During this process some nodes may have more than $b$ children, where $b$ is the branching factor. The fourth rule ensures the correct number by demoting the exceeding children to grandchildren. The last two rules propagate the correct depth to children of nodes that have been promoted or demoted. The simplicity and conceptual elegance of these rules should be compared with the parameterised recursive rule of [YM92] which uses four different kinds of path expressions and a marking command.

Some comments about this example are in order. The notation has been chosen to be as compact as possible, showing only the relevant data. Furthermore, using superscripts for depth and subscripts for children makes both of them stand out. Thus it is easier to see how a node changes from the left to the right side of a rule. A more conventional notation is obtained by translating every molecule of the form $x_c^a$ into two molecules "depth(x, a), children(x, c)", and by transforming $x$—$y$ into "linked(x, y)".

The specification can be simplified by omitting the depth of each node, as in the original example [YM92]. In fact, all that is necessary is to be able to distinguish the root node from the others. This can be done just with an extra molecule "root(x)". The first rule becomes

$$\mathrm{rc}(x),\ \mathrm{root}(x),\ x\text{—}y,\ x_{c+},\ y_{c'} \to \mathrm{rc}(x),\ \mathrm{root}(y),\ y\text{—}x,\ x_{c},\ y_{c'+}$$

the last two rules are not necessary anymore, and the superscripts of the remaining ones disappear. However, we chose to have this additional difficulty because it introduces further self-organisation.

Besides illustrating self-organisation and allowing comparison of our approach with a previous one, the example shows how structural and cardinality constraints on architectures can be specified. This is also useful for other kinds of topologies.

## 3.5 A Language

So far, our examples and those of others [IW95, IY96, IWY97] are written in an ad-hoc fashion: each uses its own syntax which may be not very uniform, the topology is sometimes not explicit, components do not have a recognisable interface, and all of the architecture is specified with a single CHAM. This last aspect was improved in [IWY98], where the authors propose each component to be specified by its own CHAM. Each component specification uses meta-variables to stand for those components or links to which it will be connected. The architecture is then the union of all CHAMs together with functions mapping the meta-variables to the actual symbols. As the authors remark, the architecture cannot be changed, since the mappings are fixed by construction.

Even so the CHAM cannot be considered an ADL, as also noticed in [Med97]. In this section we try to remedy the situation. We propose a concrete syntax for some architectural and computational abstractions and we provide a methodology to combine the description of components into an (hierarchic) architecture. Our goal is to show that with purely methodological means, without extending or changing the CHAM model, it is possible to specify the computational, structural, and interaction aspects of an architecture in a systematic way.

Our language is very simple for the moment. The rationale is to include further constructs only as necessary, when it is deemed essential for the description of dynamic architectures. Moreover, by committing just to a few design choices, it may serve as a least common ground between different existing ADLs.

There are only components and unidirectional links. The latter can be seen as wires: when there is a value on one end, it gets transferred to the other end. Unlike ACME [MGW97], we do not require two kinds of elements (components and connectors) nor

links only between different kinds of elements. This gives us freedom to describe architectures *à la* DARWIN [MK96a], which has no connectors, or C2 [OT98], which allows connectors to be linked to connectors.

A component is a molecule of the form $name{:}type{=}\{\!|state|\!\}$, where $name$ and $type$ are two identifiers, and $state$ is a solution describing the component's current state. Notice that it is encapsulated within a membrane since the state is private to each component. The state may include a set of variables, a set of ports, and, if it is a composite component, an architecture. Variables and ports are name-value pairs. Ports form a component's interface and hence may be seen as public variables: their values may be sent to and received from the environment. The value of a port can thus be changed by computation or communication. The interface itself may vary, i.e., ports may be added or deleted from a component. A variable is a molecule of the form $name{=}value$, while a port is of the form $name{\bullet}value$. When ports are within the state solution they are not visible to the environment and therefore communication cannot occur through that port. It is up to the component, through its computations, to "export" ports to the outside and "import" them back again, thus signalling when it is ready to read or send a value and when it is processing or computing a port's value, respectively.

There is a CHAM specifying the fixed elements of the language: identifiers, predefined types, connections, and message passing. The grammar of the CHAM describes the syntax of types, while the reaction rules describe the operations. For the remainder of this chapter we need the following definitions.

| | |
|---|---|
| *Name* | := *Id* ['.' *Name*] |
| *Id* | := *Letter*$^+$ [*Nat*] |
| *Letter* | := 'a' \| ... \| 'Z' |
| *Nat* | := '0' \| *Pos* |
| *NatExpr* | := *Nat* \| *Nat* mod *Pos* |
| *Pos* | := '1' [*Pos*] |
| *NatList* | := 'nil' \| *Nat* '·' *NatList* \| 'append' '(' *NatList* ',' *NatList* ')' |
| *Link* | := *Name* '—' *Name* |

We also need rules to describe the modulo and 'append' operations. The modulo is computed by reducing $n \bmod m$ until $n$ is not greater than $m$. Positive integers are written in unary notation and comparison of numbers is done with substring matching. The first rule below is for the case where the dividend is zero, while the next three rules handle positive dividends which are greater, equal, or smaller than the divisor.

$$\begin{aligned} 0 \bmod x &\to 0 \\ x1y \bmod x &\to 1y \bmod x \\ x \bmod x &\to 0 \\ x \bmod x1y &\to x \\ \mathrm{append}(n{\cdot}l,l') &\to n{\cdot}\mathrm{append}(l,l') \\ \mathrm{append}(\mathrm{nil},l) &\to l \end{aligned}$$

Regarding communication along a link, a message (i.e., molecule) $M$ is sent from port $p$ to port $q$ if they are connected, $p$ has $M$, and $q$ is ready to get it.

$$p{\bullet}M,\ p{—}q,\ q{\bullet} \to p{\bullet},\ p{—}q,\ q{\bullet}M$$

Each component type is specified by a distinct CHAM. The context-free grammar describes the actual ports, variables, and components that it contains, while the rewriting rules specify the computations and reconfigurations performed within components of that type. The grammar for a component of type *Type* (abbreviated 'T') looks like:

$$
\begin{array}{ll}
\mathit{Type} & := \mathit{Id}\ \text{':'}\ \text{'T'}\ \text{'='}\ \{\!|[\mathit{State}_T\ (\text{','}\ \mathit{State}_T)^*]|\!\} \\
\mathit{State}_T & := \mathit{Init}_T \mid \mathit{Port}_T \mid \mathit{Var}_T \mid \mathit{Comp}_T \mid \mathit{Link} \\
\mathit{Init}_T & := \ldots \\
\mathit{Port}_T & := \mathit{Id}\ \text{'}\bullet\text{'}\ \mathit{MsgType} \mid \ldots \\
\mathit{Var}_T & := \mathit{Id}\ \text{'='}\ \mathit{VarType} \mid \ldots \\
\mathit{Comp}_T & := T_1 \mid T_2 \mid \ldots \\
\mathit{Ad\text{-}hoc}_T & := \ldots
\end{array}
$$

The third production provides the available constructors for components of type 'T'. Every newly created component must be of the form t:T=$\{\!|$I$|\!\}$ where I is an initial solution generated from $\mathit{Init}_T$. Often I is a single molecule and there are reaction rules that transform it into the initial set of variables and ports, and the initial architecture (if 'T' is a composite component type).

The fourth production describes the available ports. The message type associated to each port may be described by additional productions. The fifth production is similar, but for variables.

The last two productions are only meaningful for composite components. If t:T=$\{\!|$S$|\!\}$ is such a component, these productions specify what types of component may occur in S and what molecules the user may add to S at any time. If 'T' is not a composite component type then *Link* may not appear on the right-hand side of the second production.

The reaction rules associated to component type 'T' describe its computations, interactions, and the reconfigurations that may occur within it. Hence, each component includes its own configuration manager for the subsystem it encapsulates.

We suggest that the information about a component's state and its computations is kept strictly to the necessary for the reconfiguration process, because the chemical model (as any term rewriting system) is not the most adequate to describe arbitrary computations (with sequencing, iteration, and branching). We therefore advocate to specify only those computations which change the ports' values, since programmed reconfiguration is based on the components' interfaces. The rules basically state when ports are made visible to the outside, thus constraining the interaction pattern of the component. These must be specified in such a way that whenever a port is exported, it is prefixed by the component's name, so that later the port can be imported back again.

If a component of type 'T' includes a component c of type '$T_1$', then the rules may only include molecules generated by $\mathit{Init}_{T_1}$, $\mathit{Port}_{T_1}$, and $\mathit{Ad\text{-}hoc}_{T_1}$. Other information, like the variables of c or the components it contains, are not accessible.

To make the description of the whole architecture uniform, we introduce a top-level component, called 'System' or similar, which has no ports. The total specification is then the union of all CHAMs.

As an introductory example we use a very simple system with multiple clients and a single server. The next two sections provide more elaborate examples.

A client has an output port 'req' to send requests of type *Query* and an input port 'rep' to get replies of type *Answer*. The actual types are irrelevant for our purposes. Clients are not composite components and they have no variables.

$$
\begin{array}{ll}
\mathit{Client} & := \mathit{Id}\ \text{':'}\ \text{'C'}\ \text{'='}\ \{\!|[\mathit{State}_C\ (\text{','}\ \mathit{State}_C)^*]|\!\} \\
\mathit{State}_C & := \mathit{Init}_C \mid \mathit{Port}_C \\
\mathit{Port}_C & := \text{'req'}\ \text{'}\bullet\text{'}\ \mathit{Query} \mid \text{'rep'}\ \text{'}\bullet\text{'}\ \mathit{Answer} \\
\mathit{Init}_C & := \text{'init(C)'}
\end{array}
$$

Initially, the client has generated some request.

$$\text{init(C)} \rightarrow \text{req}\bullet\text{Q, rep}\bullet$$

The client interacts with its environment exporting the 'req' port when it has a query and importing it again after the query has been sent.

$$c{:}C{=}\{\!|\mathrm{req}\bullet Q\triangleleft S|\!\} \rightarrow c.\mathrm{req}\bullet Q,\ c{:}C{=}S$$
$$c.\mathrm{req}\bullet,\ c{:}C{=}S \rightarrow c{:}C{=}\{\!|\mathrm{req}\bullet\triangleleft S|\!\}$$

The reply port has the opposite behaviour.

$$c{:}C{=}\{\!|\mathrm{rep}\bullet\triangleleft S|\!\} \rightarrow c.\mathrm{rep}\bullet,\ c{:}C{=}S$$
$$c.\mathrm{rep}\bullet A,\ c{:}C{=}S \rightarrow c{:}C{=}\{\!|\mathrm{rep}\bullet A\triangleleft S|\!\}$$

A new query is generated based on the answer to the previous one.

$$\mathrm{rep}\bullet A,\ \mathrm{req}\bullet \rightarrow \mathrm{rep}\bullet,\ \mathrm{req}\bullet Q$$

When the client wants to finish processing, it just processes the answer without generating a new query.

$$\mathrm{rep}\bullet A \rightarrow \mathrm{rep}\bullet$$

The server has for each client c a pair of ports 'req.c' and 'rep.c' to get the requests and send the replies, respectively. These ports are always visible and they are added to (removed from) the server whenever c is added to (removed from) the architecture. Although it is not relevant for reconfiguration, for illustration purposes we assume the server caches the last query processed. Initially the cache is empty.

$$\begin{array}{ll} \mathit{Server} & := \mathit{Id}\ \text{':'}\ \text{'S'}\ \text{'='}\ \{\!|[\mathit{State}_S\ (\text{','}\ \mathit{State}_S)^*]|\!\} \\ \mathit{State}_S & := \mathit{Port}_S \mid \mathit{Init}_S \mid \mathit{Var}_S \\ \mathit{Port}_S & := \mathrm{req}.\mathit{Id}\bullet\mathit{Query} \mid \mathrm{rep}.\mathit{Id}\bullet\mathit{Answer} \\ \mathit{Var}_S & := \text{'cache'}\ \text{'='}\ [\mathit{Query}\ \text{'/'}\ \mathit{Answer}] \\ \mathit{Init}_S & := \text{'cache='} \end{array}$$

The server has a single rule to answer a query and update the cache.

$$s.\mathrm{req}.c\bullet Q,\ s.\mathrm{rep}.c\bullet,\ s{:}S{=}\{\!|\mathrm{cache}{=}Q'/A'|\!\} \rightarrow s.\mathrm{req}.c\bullet,\ s.\mathrm{rep}.c\bullet A,\ s{:}S{=}\{\!|\mathrm{cache}{=}Q/A|\!\}$$

The overall system contains a single server with fixed name. The user may create and remove clients.

$$\begin{array}{ll} \mathit{Client\text{-}Server} & := \mathit{Id}\ \text{':'}\ \text{'CS'}\ \text{'='}\ \{\!|[\mathit{State}_{CS}\ (\text{','}\ \mathit{State}_{CS})^*]|\!\} \\ \mathit{State}_{CS} & := \mathit{Init}_{CS} \mid \mathit{Comp}_{CS} \mid \mathit{Link} \mid \mathit{Ad\text{-}hoc}_{CS} \\ \mathit{Comp}_{CS} & := \mathit{Client} \mid \mathit{Server} \\ \mathit{Ad\text{-}hoc}_{CS} & := \text{'cc('}\ \mathit{Id}\ \text{')'} \mid \text{'rc('}\ \mathit{Id}\ \text{')'} \\ \mathit{Init}_{CS} & := \text{'s:S=}\{\!|\mathrm{init(S)}|\!\}\text{'} \end{array}$$

The user creates a particular client-server system with a molecule of the form

$$cs{:}CS{=}\{\!|s{:}S{=}\{\!|\mathrm{init(S)}|\!\}|\!\}$$

and then adds commands to create and remove clients.

When a molecule 'cc(c)' is added to the solution, the following rule is triggered to add the component and to link it to two new ports of the server.

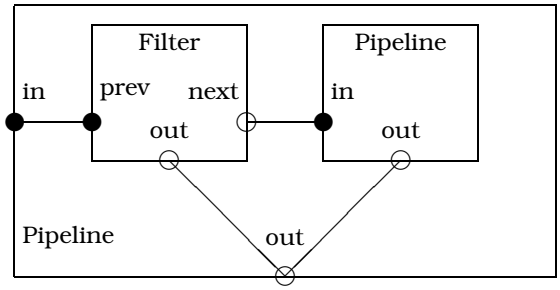
$$\mathrm{cc(c)} \rightarrow c{:}C{=}\{\!|\mathrm{init(C)}|\!\},\ c.\mathrm{req}\text{—}s.\mathrm{req}.c,\ s.\mathrm{req}.c\bullet,s.\mathrm{rep}.c\text{—}c.\mathrm{rep},\ s.\mathrm{rep}.c\bullet$$

When removing a client, the links and the respective server ports are removed again. It is important however to do it in the right moment, when no request is pending. The system configuration manager must thus look at the client's interface and wait for the ports to have no messages and to be inside the client. Looking at the client's computation rules it can be seen that in that state no new query can be generated. If the ports were outside, it would mean that a request had been sent but the reply had not been received yet.

$$\mathrm{rc(c)},\ c{:}C{=}\{\!|\mathrm{rep}\bullet\triangleleft\mathrm{req}\bullet\triangleleft S|\!\},\ c.\mathrm{req}\text{—}s.\mathrm{req}.c,\ s.\mathrm{req}.c\bullet,\ s.\mathrm{rep}.c\text{—}c.\mathrm{rep},\ s.\mathrm{rep}.c\bullet \rightarrow$$

## 3.6 Programmed Reconfiguration

To illustrate programmed reconfiguration we show how the lazy pipeline described in [MK96a] can be specified with a CHAM. The pipeline is a composite component with two communication ports called 'in' and 'out'. Internally, the pipeline consists of a filter and another pipeline. A filter has three ports: an input port 'prev' and two output ports 'next' and 'out'. The connections to the enclosing and enclosed pipelines are shown in the following diagram.




Although we use the example also to illustrate hierarchic architectures, its main characteristic is that pipelines are only created on demand, in particular, when a message is sent to its 'in' port. The example is used in [MK96a] to illustrate the use of the `dyn` keyword of the DARWIN architecture description language. We show how it is possible to obtain the same effect with the CHAM. To show a complete example, we use the lazy pipeline to implement Erasthotenes' prime number sieve. We use natural numbers: zero marks the end of input.

The syntax of the filter is given by the following grammar.

$$
\begin{array}{ll}
\mathit{Filter} & := \mathit{Id}\ \text{':'}\ \text{'F'}\ \text{'='}\ \{\!|[\mathit{State}_F\ (\text{','}\ \mathit{State}_F)^*]|\!\} \\
\mathit{State}_F & := \mathit{Port}_F \mid \mathit{Init}_F \mid \mathit{Var}_F \\
\mathit{Port}_F & := \text{'prev'}\ \text{'•'}\ [\mathit{Pos}] \mid \text{'next'}\ \text{'•'}\ [\mathit{Pos}] \mid \text{'out'}\ \text{'•'}\ [\mathit{Pos}] \\
\mathit{Var}_F & := \text{'state'}\ \text{'='}\ (\text{'new'} \mid \text{'ready'} \mid \text{'check'}) \mid \text{'prime'}\ \text{'='}\ \mathit{Pos} \mid \text{'test'}\ \text{'='}\ \mathit{NatExpr} \\
\mathit{Init}_F & := \text{'init(F)'}
\end{array}
$$

Initially, the ports are inside the filter and have no values.

$$\text{init(F)} \rightarrow \text{prev•, next•, out•, state=new}$$

The computations performed by the filter are as follows. First it exports the 'prev' port and imports it again only when there is some input. For simplicity we assume in this example that variables $n$, $p$, $x$, and $y$ only match against positive numbers.

$$
\begin{array}{c}
f{:}F{=}\text{prev•}\triangleleft S \rightarrow f.\text{prev•},\ f{:}F{=}S \\
f.\text{prev•}n,\ f{:}F{=}S \rightarrow f{:}F{=}\text{prev•}n\triangleleft S
\end{array}
$$

If it is the first input, it must be a new prime number. It is stored in variable 'prime' and sent to the 'out' port, which behaves in the opposite way of 'prev'.

$$
\begin{array}{c}
\text{state=new, prev•}n,\ \text{out•} \rightarrow \text{state=ready, prime=}n,\ \text{prev•, out•}n \\
f{:}F{=}\text{out•}n\triangleleft S \rightarrow f.\text{out•}n,\ f{:}F{=}S \\
f.\text{out•},\ f{:}F{=}S \rightarrow f{:}F{=}\text{out•}\triangleleft S
\end{array}
$$

If the number $n$ is not the first input, then it must be checked whether it is a multiple of the stored prime number $p$, i.e., if $n$ mod $p$ is zero. In that case the number $n$ is ignored. Otherwise, it may be a prime number, and thus it is sent to the next filter through port 'next' which behaves like 'out'.

$$\begin{aligned}
\text{state=ready, prev}\bullet n, \text{ prime=}p &\to \text{state=check, prev}\bullet n, \text{ prime=}p, \text{ test=}n \bmod p\\
\text{state=check, prev}\bullet n, \text{ test=}0 &\to \text{state=ready, prev}\bullet\\
\text{state=check, prev}\bullet n, \text{ next}\bullet, \text{ test=}1x &\to \text{state=ready, prev}\bullet, \text{ next}\bullet n\\
f\text{:F=next}\bullet n \triangleleft S &\to f\text{.next}\bullet n, \ f\text{:F=}S\\
f\text{.next}\bullet, \ f\text{:F=}S &\to f\text{:F=next}\bullet \triangleleft S
\end{aligned}$$

We now turn to the pipeline.

$$\begin{aligned}
\textit{Pipeline} &:= \textit{Id}\ \text{':'}\ \text{'P'}\ \text{'='}\ \{\!|[\textit{State}_P\ (\text{','}\ \textit{State}_P)^*]|\!\}\\
\textit{State}_P &:= \textit{Port}_P \mid \textit{Var}_P \mid \textit{Init}_P \mid \textit{Comp}_P \mid \textit{Link}\\
\textit{Port}_P &:= \text{'in'}\ \text{'}\bullet\text{'}\ [\textit{Pos}] \mid \text{'out'}\ \text{'}\bullet\text{'}\ [\textit{Pos}]\\
\textit{Var}_P &:= \text{'dyn'}\ \text{'='}\ \textit{Name}\ \text{'/'}\ \textit{Molecule} \mid \text{'pipeline'}\ \text{'='}\ (\text{'yes'} \mid \text{'no'})\\
\textit{Comp}_P &:= \textit{Filter} \mid \textit{Pipeline}\\
\textit{Init}_P &:= \text{'init(P)'}
\end{aligned}$$

Initially the pipeline contains its ports, a filter partially connected, and the indication that the filter's 'next' port dynamically creates a new pipeline.

$$\begin{aligned}
\text{init(P)} \to\ & \text{in}\bullet, \text{ out}\bullet, \text{ f:F=}\{\!|\text{init(F)}|\!\}, \text{ in—f.prev, f.out—out,}\\
& \text{pipeline=no, dyn=f.next/}\{\!|\text{pipeline=yes, f.next—p.in, p.out—out}|\!\}
\end{aligned}$$

When the filter sends its first message through port 'next', a new pipeline and its connections are created by the following rule which allows to attach an arbitrary solution to a port and add that solution to the system once the first message is received by or sent from that port. The solution may contain arbitrary reconfiguration commands, new components and connections. It thus generalises DARWIN's dyn construct.

$$p\bullet msg, \text{ dyn=}p/\{\!|S|\!\} \to p\bullet msg, \ S$$

We must of course make sure that the solution evolves only when "freed" from its membrane. In our case, only conflicting information about the existence of a pipeline will create a new one.

$$\text{pipeline=no, pipeline=yes} \to \text{pipeline=yes, p:P=}\{\!|\text{init(P)}|\!\}$$

We have thus programmed reconfiguration and the recursive structure of the architecture is immediately apparent from the grammar and from the existence of 'init(P)' on both sides of rules.

## 3.7 A Mixed Example

This section presents a single, compact example to illustrate how the various approaches might be represented using the CHAM formalism. As several kinds of evolution are mingled, the CHAM to be presented is not very uniform, nor the simplest possible.

The chosen example is a client-server system where the server is a printing system consisting of a spooler and a set of printers. Each client makes a single request to print a document. Requests are queued in FIFO order by the spooler. The spooler selects randomly an available printer to print the next document. Printers may break down while printing. If no printer is working then any new client requests are immediately rejected.

The chosen architecture consists of zero or more clients, one name server, one "inhibitor" and one composite component containing a spooler and zero or more printers. If there are no (working) printers, the name server is linked to the "inhibitor" otherwise it is linked to to the printing system. Whenever a new client arrives, it is automatically connected to the name server, whose name is known and whose job is to relink the client to the component the name server is linked to. The "inhibitor" just rejects any client

request. The printing system accepts and queues any incoming request. Clients automatically leave the system after getting a (positive or negative) reply. Printers must be explicitly removed (when broken) and introduced (e.g., after repair or purchase). When the last working printer breaks down, the name server is unlinked from the printing system and linked to the "inhibitor". Inversely, if no printer is working and a new one is added, the name server is linked back to the printing system.

We start with the name server. It has two ports but they are not used for communication.

$$\begin{array}{lll} \mathit{NameServer} & := \mathit{Id}\ \text{':'}\ \text{'N'}\ \text{'='}\ \{\!|[\mathit{State}_N\ (\text{','}\ \mathit{State}_N)^*]|\!\} \\ \mathit{State}_N & := \mathit{Port}_N \\ \mathit{Port}_N & := \text{'cl'}\ \text{'}\bullet\text{'}\ \mid \text{'sv'}\ \text{'}\bullet\text{'} \end{array}$$

Hence a name server is always created as $n{:}N{=}\{\!|\}$. All it does is simply to relink a client to the current server (either the inhibitor or the printing system) via a bidirectional connection.

$$c\text{—}n.cl,\ n.sv\text{—}s \rightarrow c\text{—}s,\ s\text{—}c,\ n.sv\text{—}s$$

The client has a single port 'rr' to accept requests and send replies. Initially the port is inside the client and its value is the number of pages to print.

$$\begin{array}{lll} \mathit{Client} & := \mathit{Id}\ \text{':'}\ \text{'C'}\ \text{'='}\ \{\!|[\mathit{State}_C\ (\text{','}\ \mathit{State}_C)^*]|\!\} \\ \mathit{State}_C & := \mathit{Port}_C \mid \mathit{Init}_C \\ \mathit{Port}_C & := \text{'rr'}\ \text{'}\bullet\text{'}\ (\ \mathit{Pos} \mid \text{'accepted'} \mid \text{'rejected'}) \\ \mathit{Init}_C & := \text{'rr'}\ \text{'}\bullet\text{'}\ \mathit{Pos} \end{array}$$

The following three rules describe the behavior of a client. The first one links it to the name server (thereby exporting the port to signal it is ready to communicate and in order to avoid duplicate connections) and the other two remove the client and its connection after getting a reply.

$$\begin{array}{r} c{:}C{=}\{\!|rr\bullet p|\!\},\ n{:}N{=}S \rightarrow c.rr\bullet p,\ c{:}C{=}\{\!|\},\ n{:}N{=}S,\ c.rr\text{—}n.cl \\ c.rr\bullet rejected,\ c{:}C{=}\{\!|\},\ c.rr\text{—}s,\ s\text{—}c.rr \rightarrow \\ c.rr\bullet accepted,\ c{:}C{=}\{\!|\}\ c.rr\text{—}s,\ s\text{—}c.rr \rightarrow \end{array}$$

The inhibitor also has an input/output port.

$$\begin{array}{lll} \mathit{Inhibit} & := \mathit{Id}\ \text{':'}\ \text{'I'}\ \text{'='}\ \{\!|[\mathit{State}_I\ (\text{','}\ \mathit{State}_I)^*]|\!\} \\ \mathit{State}_I & := \mathit{Port}_I \mid \mathit{Init}_I \\ \mathit{Port}_I & := \text{'rr'}\ \text{'}\bullet\text{'}\ [\mathit{Pos} \mid \text{'rejected'}] \\ \mathit{Init}_I & := \text{'rr}\bullet\text{'} \end{array}$$

The port is first exported and then it replies to any incoming request with a rejection.

$$\begin{array}{r} i{:}I{=}\{\!|rr\bullet|\!\} \rightarrow i.rr\bullet,\ i{:}I{=}\{\!|\} \\ i.rr\bullet 1n \rightarrow i.rr\bullet rejected \end{array}$$

A printer has two public ports to show to the print server the state and the number of pages that remain to print.

$$\begin{array}{lll} \mathit{Printer} & := \mathit{Id}\ \text{':'}\ \text{'P'}\ \text{'='}\ \{\!|[\mathit{State}_P\ (\text{','}\ \mathit{State}_P)^*]|\!\} \\ \mathit{State}_P & := \mathit{Port}_P \mid \mathit{Init}_P \\ \mathit{Port}_P & := \text{'pages'}\ \text{'}\bullet\text{'}\ [\mathit{Nat}] \mid \text{'state'}\ \text{'}\bullet\text{'}\ (\text{'ok'} \mid \text{'broken'}) \\ \mathit{Init}_P & := \text{'init(P)'} \end{array}$$

Initially the printer is working and there are no pages to print.

$$init(P) \rightarrow state\bullet ok,\ pages\bullet 0$$

The printer starts by exporting the ports.

$$p{:}P{=}\{\!|pages{\bullet}0,\ state{\bullet}ok|\!\} \to p.pages{\bullet}0,\ p.state{\bullet}ok,\ p{:}P{=}\{\!|\,|\!\}$$

Each printer outputs a document one page at a time. When reaching the end, it becomes available again. However, while printing a page, the printer may break down.

$$\begin{aligned} p.pages{\bullet}1,\ p.state{\bullet}ok &\to p.pages{\bullet}0,\ p.state{\bullet}ok \\ p.pages{\bullet}1{:}1n,\ p.state{\bullet}ok &\to p.pages{\bullet}1n,\ p.state{\bullet}ok \\ p.pages{\bullet}1n,\ p.state{\bullet}ok &\to p.pages{\bullet}1n,\ p.state{\bullet}broken \end{aligned}$$

The print server has a single port to communicate with clients. It has a counter to keep track of how many printers are working. It also contains variables to store the queue of requests, and the commands to connect and disconnect from the name server. Users may create and remove printers.

| | |
|---|---|
| *PrintServer* | := *Id* ':' 'S' '=' {\|[*State*$_S$ (',' *State*$_S$)$^*$]\|} |
| *State*$_S$ | := *Port*$_S$ \| *Init*$_S$ \| *Var*$_S$ \| *Comp*$_S$ \| *Link* \| *Ad-hoc*$_S$ |
| *Port*$_S$ | := 'rr' '•' (*Pos* \| 'accepted' \| *Rule*) |
| *Var*$_S$ | := 'working' '=' *Nat* \| 'queue' '=' *NatList* \| ('link' \| 'unlink') '=' *Rule* |
| *Rule* | := *Solution* '↦' *Solution* |
| *Init*$_S$ | := 'init(S,' *Nat* ',' *Id* ',' *Id* ')' |
| *Comp*$_S$ | := *Printer* |
| *Ad-hoc*$_S$ | := 'cc(' *Id* ')' \| 'rc(' *Id* ')' |

To create a print server it is necessary to specify the number of printers it contains initially. It is also necessary to give the ports of the print server, the name server and the inhibitor. This allows to build in advance the rules necessary to (dis)connect the print server and the name server.

$$\begin{aligned} init(S,\ 0,\ s,\ n,\ i) &\to rr{\bullet},\ queue{=}nil,\ working{=}0, \\ &\qquad unlink{=}\{\!|n{—}s|\!\}{\mapsto}\{\!|n{—}i|\!\},\ link{=}\{\!|n{—}i|\!\}{\mapsto}\{\!|n{—}s|\!\} \\ init(S,\ 1{:}1x,\ s,\ n,\ i) &\to init(S,\ 1x,\ s,\ n,\ i),\ cc(p) \\ init(S,\ 1,\ s,\ n,\ i) &\to init(S,\ 0,\ s,\ n,\ i),\ cc(p) \end{aligned}$$

Any incoming request is put at the end of the spooler queue and replied with an acceptance. The spooler removes requests from the beginning of the queue as long as there is an available printer.

$$\begin{aligned} rr{\bullet}1n,\ queue{=}q &\to rr{\bullet}accepted,\ queue{=}append(q,\ 1n{\cdot}nil) \\ p.pages{\bullet}0,\ p.state{\bullet}ok,\ queue{=}n{\cdot}q &\to p.pages{\bullet}n,\ p.state{\bullet}ok,\ queue{=}q \end{aligned}$$

When a printer breaks down, the counter is decremented. If it is the last working printer, then the name server must get linked to the "inhibitor". To that end, the printing system sends a message representing the reaction rule that replaces the existing connection by the new one.

$$p.pages{\bullet}1n,\ p.state{\bullet}broken,\ working{=}1{:}1n \to p.pages{\bullet}0,\ p.state{\bullet}broken,\ working{=}1n$$

$$p.pages{\bullet}1n,\ p.state{\bullet}broken,\ working{=}1,\ rr{\bullet},\ unlink{=}r \to p.pages{\bullet}0,\ p.state{\bullet}broken,\ working{=}0,\ rr{\bullet}r,\ unlink{=}r$$

Broken printers only "disappear" from the architecture when they are explicitly taken away for repair, and new printers must also be added explicitly. This is done by dropping the molecules given in *Ad-hoc*$_S$ into the subsolution representing the printing system. If there were no working printers, the addition of a new one removes the connection between the name server and the inhibitor and creates a connection to the printing system.

$$\text{rc}(p),\ p.\text{pages}\bullet 0,\ p.\text{state}\bullet\text{broken},\ p{:}P{=}S \rightarrow$$
$$\text{cc}(p),\ \text{working}{=}0,\ \text{rr}\bullet,\ \text{link}{=}r \rightarrow p{:}P{=}\{\!|\text{init}(P)|\!\},\ \text{working}{=}1,\ \text{rr}\bullet r,\ \text{link}{=}r$$
$$\text{cc}(p),\ \text{working}{=}1n \rightarrow p{:}P{=}\{\!|\text{init}(P)|\!\},\ \text{working}{=}11n$$

Since the 'rr' port is used both to receive input from the clients as well as send reconfiguration rules to the outside, it must be able to cross the membrane freely.

$$p.\text{rr}\bullet msg,\ p{:}P{=}S \leftrightarrow p{:}P{=}\{\!|\text{rr}\bullet msg \triangleleft S|\!\}$$

The whole system has the following syntax, allowing the user to add clients.

$$\begin{array}{ll} \textit{System} & := \textit{Id}\ \text{':'}\ \text{'Sys'}\ \text{'='}\ \{\![\textit{State}_{\text{Sys}}\ (\text{','}\ \textit{State}_{\text{Sys}})^*]\!\} \\ \textit{State}_{\text{Sys}} & := \textit{Init}_{\text{Sys}} \mid \textit{Comp}_{\text{Sys}} \mid \textit{Link} \mid \textit{Ad-hoc}_{\text{Sys}} \\ \textit{Init}_{\text{Sys}} & := \text{'Init(Sys,'}\ \textit{Nat}\ \text{')'} \\ \textit{Comp}_{\text{Sys}} & := \textit{Client} \mid \textit{NameServer} \mid \textit{PrintServer} \mid \textit{Inhibit} \\ \textit{Ad-hoc}_{\text{Sys}} & := \textit{Id}{:}C{=}\{\!|\textit{Init}_C|\!\} \end{array}$$

Initially the inhibitor, the name server, and the print server are created with fixed names. The name server is linked to the inhibitor. The number of printers is given by the user. When the first one is created, the name server will be connected to the print server.

$$\text{init}(S,\ n) \rightarrow i{:}I{=}\{\!|\text{rr}\bullet|\!\},\ n{:}N{=}\{\!|\,|\!\},\ s{:}S{=}\{\!|\text{init}(P,\ n,\ s.\text{rr},\ n.\text{sv},\ i.\text{rr})|\!\},\ n.\text{sv—i.rr}$$

A message representing a reaction is "executed" when it is in a context that satisfies the left-hand side of the reaction. The following general rule expresses this. It can be seen as a special kind of CHAM meta-interpreter.

$$S,\ p\bullet\{\!|S|\!\} \mapsto \{\!|S'|\!\} \rightarrow S',\ p\bullet$$

To see how this CHAM works let us work through a concrete example, seeing how the solution that describes the architecture and the state of each component evolves. The initial solution states that the system has one printer.

$$\text{sys}{:}\text{Sys}{=}\{\!|\text{init}(\text{Sys},\ 1)|\!\}$$

The constructor generates the initial components.

$$\text{sys}{:}\text{Sys}{=}\{\!|i{:}I{=}\{\!|\text{rr}\bullet|\!\},\ n{:}N{=}\{\!|\,|\!\},\ n\text{—i},\ s{:}S{=}\{\!|\text{init}(S,\ 1,\ s.\text{rr},\ n.\text{sv},\ i.\text{rr})|\!\}|\!\}$$

All computations and reconfigurations occur within 'sys'. Therefore we henceforth only present the encapsulated solution to make the example easier to follow.

The print server's constructor starts generating its ports, variables and the enclosed printer (with a random name), while the inhibitor exports its port.

$$i.\text{rr}\bullet,\ i{:}I{=}\{\!|\,|\!\},\ n{:}N{=}\{\!|\,|\!\},\ n.\text{sv—i.rr},\ s{:}S{=}\{\!|\text{init}(S,\ 0,\ s.\text{rr},\ n.\text{sv},\ i.\text{rr}),\ \text{cc}(x)|\!\}$$

While the print server continues the initialisation phase, the user adds a client with a request to print a three page document. The client is added directly (not through a creation command) because of production $\textit{Ad-hoc}_{\text{Sys}}$.

$$\begin{array}{c} c{:}C{=}\{\!|\text{rr}\bullet 111|\!\},\ i.\text{rr}\bullet,\ i{:}I{=}\{\!|\,|\!\},\ n{:}N{=}\{\!|\,|\!\},\ n.\text{sv—i.rr}, \\ s{:}S{=}\{\!|\text{rr}\bullet,\ \text{queue}{=}\text{nil},\ \text{unlink}{=}\{\!|n.\text{sv—s.rr}|\!\}{\mapsto}\{\!|n.\text{sv—i.rr}|\!\},\ \text{link}{=}\{\!|n.\text{sv—i.rr}|\!\}{\mapsto}\{\!|n.\text{sv—s.rr}|\!\}, \\ \text{working}{=}0,\ \text{cc}(x)|\!\} \end{array}$$

The client is linked to the name server and the print server creates the printer and prepares to send the reconfiguration rule.

$$\begin{array}{c} c.\text{rr}\bullet 111,\ c{:}C{=}\{\!|\,|\!\},\ c.\text{rr—n.cl},\ i.\text{rr}\bullet,\ i{:}I{=}\{\!|\,|\!\},\ n{:}N{=}\{\!|\,|\!\},\ n.\text{sv—i.rr}, \\ s{:}S{=}\{\!|\text{rr}\bullet\{\!|n.\text{sv—i.rr}|\!\}{\mapsto}\{\!|n.\text{sv—s.rr}|\!\},\ \text{queue}{=}\text{nil},\ \text{unlink}{=}\{\!|n.\text{sv—s.rr}|\!\}{\mapsto}\{\!|n.\text{sv—i.rr}|\!\}, \\ \text{link}{=}\{\!|n.\text{sv—i.rr}|\!\}{\mapsto}\{\!|n.\text{sv—s.rr}|\!\},\ \text{working}{=}1,\ x{:}P{=}\{\!|\text{init}(P)|\!\}|\!\} \end{array}$$

The client is relinked to the name server and the rule is exported. The printer is initialised. Henceforth we omit the 'link' and 'unlink' variables because their values do not change.

c.rr•111, c:C={|}, c.rr—i.rr, i.rr—c.rr, i.rr•, i:I={|}, n:N={|}, n.sv—i.rr,
s.rr•{|n.sv—i.rr|}↦{|n.sv—s.rr|}, s:S={|queue=nil, working=1, x:P={|state•ok, pages•0|}|}

Using the general communication rule, the client's request is passed to the inhibitor. The reconfiguration rule is applied, and the printer makes its state public.

c.rr•, c:C={|}, c.rr—i.rr, i.rr—c.rr, i.rr•111, i:I={|}, n:N={|}, n.sv—s.rr,
s.rr•, s:S={|queue=nil, working=1, x.state•ok, x.pages•0, x:P={|}|}

The inhibitor rejects the request, sends back the reply to the client and the latter goes away. Meanwhile the user adds a new client, which gets linked to the print server. In the following sequence we omit the print server because its state does not change.

c.rr•, c:C={|}, c.rr—i.rr, i.rr—c.rr, i.rr•reject, i:I={|}, n:N={|}, n.sv—s.rr, d:C={|rr•11|}

c.rr•reject, c:C={|}, c.rr—i.rr, i.rr—c.rr, i.rr•, i:I={|}, n:N={|}, n.sv—s.rr, d.rr•11, d:C={|},
d.rr—n.cl

i.rr•, i:I={|}, n:N={|}, n.sv—s.rr, d.rr•11, d:C={|}, d.rr—s.rr, s.rr—d.rr

The request is passed to the print server port.

i.rr•, i:I={|}, n:N={|}, n.sv—s.rr, d.rr•, d:C={|}, d.rr—s.rr, s.rr—d.rr,
s.rr•11, s:S={|queue=nil, working=1, x.state•ok, x.pages•0, x:P={|}|}

We do not further use the inhibitor and the name server in our example and hence omit them henceforth. The request enters the printing system,

d.rr•, d:C={|}, d.rr—s.rr, s.rr—d.rr,
s:S={|rr•11, queue=nil, working=1, x.state•ok, x.pages•0, x:P={|}|}

is accepted and queued,

d.rr•, d:C={|}, d.rr—s.rr, s.rr—d.rr,
s:S={|rr•accept, queue=append(nil,11·nil), working=1, x.state•ok, x.pages•0, x:P={|}|}

and is processed while the reply is sent back to the client, which then goes away.

d.rr•, d:C={|}, d.rr—s.rr, s.rr—d.rr,
s.rr•accepted, s:S={|queue=11·nil, working=1, x.state•ok, x.pages•0, x:P={|}|}

d.rr•accepted, d:C={|}, d.rr—s.rr, s.rr—d.rr,
s.rr•, s:S={|queue=nil, working=1, x.state•ok, x.pages•11, x:P={|}|}

s.rr•, s:S={|queue=nil, working=1, x.state•ok, x.pages•1, x:P={|}|}

The printer breaks down while trying to print the second page.

s.rr•, s:S={|queue=nil, working=1, x.state•broken, x.pages•1, x:P={|}|}

The user adds a command to remove it and the print server copies the rule stored in the 'unlink' variable (shown previously) to the port.

s:S={|rr•, queue=nil, working=1, rc(x), x.state•broken, x.pages•1, x:P={|}|}

s:S={|rr•{|n.sv—s.rr|}↦{|n.sv—i.rr|}, queue=nil, working=0, rc(x), x.state•broken,
x.pages•0, x:P={|}|}

$$s.rr \bullet \{\!|n.sv—s.rr|\!\} \mapsto \{\!|n.sv—i.rr|\!\},\ s{:}S = \{\!|queue{=}nil,\ working{=}0|\!\}$$

The rule would now be applied to disconnect the name server from the print server and to connect it back to the inhibitor, and the user would have to add create commands to have working printers. We would thus get the same situation as at the beginning.

Notice that when the first working printer was created in the initialisation of the system, the name server is not immediately (i.e., in the same step) relinked to the print server. Thus the first client had its request rejected although there was an idle printer. This is quite realistic for this architecture: there is a separate component to reject requests and hence it is natural that it takes some time to propagate the new state of the printing system. This deals with one of the issues raised in [Ore98]: the behavioural view of an architecture should reflect the fact that in reality reconfigurations are not performed atomically by the operating system. Therefore the intermediate states of a system during change may be somehow inconsistent. In our example, inconsistency means that the topology of the architecture (i.e., which component the name server is linked to) does not always reflect the state of the printing component (i.e., whether it has working printers or not). However, after the change has taken place, topology and system state are consistent again.

To summarise, the example has illustrated

**self-organisation** the clients are automatically linked to and unlinked from the correct component: either the "inhibitor" or the printing system;

**ad-hoc reconfiguration** printers are explicitly removed and created by reconfiguration commands that are added unpredictably to the solution;

**programmed reconfiguration** the connection between the name server is switched from the "inhibitor" to the printing system and vice-versa depending on the number of working printers;

**code mobility** the programmed reconfiguration is achieved by sending the "program" (in this case a single rule) from within the printing system to the outside;

**interaction** computations affect the way the architecture changes (e.g., in programmed reconfiguration) and reconfigurations trigger computations (e.g., the addition of a new client);

**parallel reconfiguration** changes within the printing system (e.g., printer removal) and outside it (e.g., client creation) can occur simultaneously;

**non-atomic changes** the name server connection is not changed atomically with the creation of the first working printer or with the breaking of the last one.

## 3.8 Concluding Remarks

Inverardi and others [IW95, IY96, IWY97] have shown that the Chemical Abstract Machine [BB92] is a useful tool to describe and study the computational behaviour of a system with a static architecture. In this chapter we extended the work in three directions. First, to handle not a single architecture but whole classes of architectures, by specifying a style as a non-deterministic self-organised reconfiguration process starting with an "empty" architecture. Second, to cope with dynamic architectures; in particular, we dealt with self-organised, ad-hoc, and programmed reconfiguration, global and local structural constraints, and a restricted form of code mobility, by encoding rules (but not rule schemata) as molecules. Third, we have made a first attempt at a minimal CHAM-based architecture description language. Based on our preliminary exploration,

we conclude that the CHAM may be used for the specification of software architecture style and reconfiguration due to following characteristics of the chemical model.

**simplicity** There is a single data structure (multiset of terms) and a single programming construct ("if-then" rewrite rules), both of which are familiar and intuitive concepts. Also, there are only two constructs for modularisation (membranes and airlocks), allowing the description of hierarchical architectures with encapsulated state. Furthermore, although data is local to each composite component, rules are global, allowing the designer to describe computations and reconfigurations that are common to several (composite) components. For example, a single rule was used to describe message passing among arbitrary components.

**suitability** The model's view of "computation as the global evolution of a collection of atomic values interacting freely" [BM96] is naturally suited to describe the evolution of self-organising architectures. Also, the combination of reaction rules (the interactions) and initial solution of molecules (the atomic values) can be used to specify both system and node integrity properties [YM92]. Reaction rules are inherently operational and parallel in nature, and are triggered upon the presence of some particular molecules in the solution, given on the left-hand side of rules. This means that the CHAM describes modifications and constraints, and that it is particularly useful for architectures which are highly dynamic (e.g., if there are infinite distinct configurations, something an approach like [ADG98] cannot handle) and whose reconfiguration is self-organised or ad-hoc and depends on local properties.

**flexibility** The definition of the molecules is left to the designer. This has a twofold advantage. First, as much or as little detail can be included as necessary for the application at hand (e.g., clients interact with the spooler through explicit connections, and messages, while the communication between the print server and printers is implicit through their current state). Second, a molecule can represent an element of the architecture (i.e., a composite component or a connection), a reconfiguration command or reaction rule, or auxiliary data (like counters). As for components, it is possible to represent their structural and computational aspects. The former include the connections a component has, the latter describe the component's state. As for reconfiguration commands, they may be high-level (like the 'cc' command which also creates new links) or low-level ones (like the 'create' command). With all these options one can encode and compare several models of reconfiguration described in the literature within a uniform framework. With the CHAM, the diverse approaches are easily recognisable according to where molecules that represent reconfigurations (like 'cc', 'dyn=. . . /. . . ', and '$\{\!|\ldots|\!\}\mapsto\{\!|\ldots|\!\}$') appear in the reaction rules: in programmed reconfiguration they appear on both sides, in ad-hoc reconfiguration they appear only in left-hand sides, and in self-organisation they do not appear at all.

The small number of constructs provided by the CHAM model also has disadvantages. Since molecules are entirely built from the constants and operators defined by the user, architectures which need common data types (like booleans and integers) and operations (like sum and comparison) are a bit tedious and lengthy to specify, and not very readable. Flexibility may be problematic too. As seen in previous sections, each CHAM introduces its own syntax and assumptions. This requires additional explanations and does not allow reuse of specifications. Moreover, the CHAM lacked the syntactic constructs to serve as an ADL, i.e., to represent architectures in a modular way with explicit connections between component interfaces. We have shown that it is possible to achieve that goal without resorting to additional formalisms: we chose a fixed representation for the most important concepts (like component and connection) and imposed some constraints on the grammars used to describe the molecules' syntax. Our position is thus

that the CHAM may be used not only as a theoretical framework but also as a practical specification language, and the chapter presented one such proto-ADL. By including only a very small set of constructs we wish to contribute to a wider discussion on what are the minimal foundations for dynamic architectures and their description languages on which higher-level abstractions can be built.

There are many possibilities for future work. One of them is to investigate the means to provide support for partial mechanical analysis of complex properties and architectures. There are several potential independent lines of action.

- A CHAM is similar to term rewriting system (TRS) with an associative and commutative operator: multiset union. It might be possible to use or adapt techniques developed for TRS to prove termination of reconfiguration and uniqueness of the resulting architecture.
- The CHAM can be encoded in rewriting logic [Mes96] and thus tools for that framework, like Maude [CELM96] and ELAN [BKK+96], could be used to test architecture specifications written in CHAM. Another possibility is to adapt and expand sequential and parallel implementations of Gamma, the original chemical model [Cre91, BCM88].
- A third kind of approach could be based on temporal logic and model-checking as described in [CFM98, CI99].
- If the style is described by a context-free grammar, the algorithm of [Mét98] can be applied to check whether a given reconfiguration graph grammar keeps the given style. It remains to be seen if the method can be adapted to the general CHAM model or if a new algorithm can be developed.

Another avenue for future work is to extend the proposed language with constructs that make specifications easier to write. For example, PoliS [CFM98] and its successor MobiS [Mas99] allow to tag molecules on the left-hand side of rules to indicate they are to be only read, not consumed, thus avoiding the need to repeat them on the right-hand side. They also allow rules to describe local computations of values, thus providing a way to use common data types (like integers) directly, without resorting to additional rules. The languages include further primitives to handle mobility, like an operator to consume or produce a molecule in the enclosing solution instead of the "current" one. It will be interesting to investigate which of these constructs cannot be described using the pure CHAM.

We are of course aware that the chemical model is not suited for every kind of style or reconfiguration. Since a reaction depends on the presence of some molecules, reconfigurations that depend on "negative" or global conditions (e.g., "if every client is not connected to the printing system, then. . . ") may be impossible or very difficult to specify, leading to CHAMs that are cumbersome to write and hard to understand. We also expect several global integrity constraints to be not as easy to express as the branching factor or the depth of a tree. However, we hope that our exploration has reinforced the suggestion that "the CHAM model might be one useful tool in the software architect's chest of useful tools" [IW95].

# Chapter 4

# The CommUnity Approach

CommUnity is a parallel program design language initially developed by José Fiadeiro and Tom Maibaum [FM95] to show how programs fit into Goguen's categorical approach to General Systems Theory. It is an action based version of Unity [CM88], but it also draws elements from IP [FF96]. Since then, the language and its framework have been extended to provide a formal platform for architectural design of open, reactive, reconfigurable systems [FM96, Fia96, FM97, FL97, WF98b, WF98a, Lop99, LF99, WF99].

We gradually present the syntax and semantics of CommUnity and show how a function mapping the vocabulary of two programs is able to capture superposition [Kat93]. That function is called a program morphism. A configuration is then a graph with nodes labelled by programs and arcs labelled by morphisms. Each configuration can be transformed into a single, semantically equivalent, program that represents the whole system. The transformation operation is given by a universal categorical construction called colimit. A connector is a particular case of configuration and we present several fundamental connectors (like message passing). A transient connector has an associated condition to state when it is active. An architecture is then a configuration built from (transient) connectors and individual programs. Hence a static graph is able to represent reconfigurations due to transient interactions.

To represent more general reconfigurations we use labelled graph rewriting. To stay within a uniform framework, we adopt the algebraic approach to graph transformation [CMR+96a]. To cater for run-time reconfiguration, we add to the node labels the usual representation of state as pairs of variables and values, and graph rewriting rules become conditional. Dynamic reconfiguration is then the interleaving of rewritings performed on the architecture and computations performed on the colimit.

Finally, we add the notion of architectural style, given by a fixed graph. An architecture conforms to a style if there is a structure preserving mapping from the former to the latter. The style's purpose is to constrain the possible connections between components, allowing the use of COTS (connectors off the shelf). The categorical framework guarantees that if the initial architecture and each rewriting rule conforms to a given style, so does the resulting architecture.

The mathematical concepts needed for this approach are defined in Chapter A on page 109.

## 4.1 Example

The running example to be used in this chapter is inspired in the airport luggage distribution system used to illustrate Mobile Unity [RMP97]. One or more carts move continuously on a $U$ units long circular track. A cart advances one unit at each step. All carts move in the same direction.

Along the track there are stations. There is at most one station per unit. Each station corresponds to a check-in counter or to a gate. Carts take bags from check-in stations to gate stations. All bags from a given check-in go to the same gate. A cart transports at most one bag at a time. When it is empty, the cart picks a bag up from the nearest check-in.

Carts must not bump into each other. This could happen if a cart is moving and the cart in front of it is stopped at a station loading or unloading a bag.

As an example of an ad-hoc reconfiguration we consider that management decides to equip each cart with two counters to compute how many bags are processed on average for each completed lap. The rationale is to check how efficient is the track layout, i.e., the distribution of the stations along the circuit.

## 4.2 Types and Expressions

This section presents the necessary concepts for COMMUNITY programs to represent and manipulate data: types, operators, variables, and the expressions that can be built from them. It also shows how variables from different programs can be related and hence expressions can be translated.

### 4.2.1 Syntax

COMMUNITY is independent of the actual data types used. Therefore it assumes there are pre-defined types and functions given by a fixed signature in the usual algebraic sense [EM85].

**Definition 4.1.** An *algebraic signature* is a tuple $\langle S, F \rangle$ where

- $S$ is a set of *sort symbols*,
- $F$ is an $S^* \times S$-indexed family of sets of *function symbols*.

All these sets are finite and mutually disjoint. □

The functions are defined by a set of equational [EM85] or first-order [MVS85] axioms.

**Definition 4.2.** An *algebraic data type specification* is a tuple $\langle S, F, \Phi \rangle$ where

- $\langle S, F \rangle$ is an algebraic signature,
- $\Phi$ is a set of *axioms*. □

Having fixed the types, we may proceed to expressions. We start with the definition of variables, which must be typed.

**Definition 4.3.** A set of *variables* is a set $X = \bigcup_{s \in S} X_s$ such that sets $X_s$ are mutually disjoint. □

COMMUNITY has three kinds of expression: terms, defined from functions and variables as usual, set expressions, each denoting a set of terms of the same type, and propositions, which have a truth value. To make the formalisation easier, the languages to be introduced are minimal. Other constructs (like set intersection and logical disjunction) can be defined as abbreviations.

**Definition 4.4.** Let $X$ be a set of variables. The language $Terms_s(X)$ of *terms* of sort $s$ is defined by

$$t_s := v \mid c \mid f(t_{s_1}, \ldots, t_{s_n})$$

where $v \in X_s$, $c \in F_{\langle\rangle,s}$, and $f \in F_{\langle s_1,\ldots,s_n\rangle,s}$. The language $Sets_s(X)$ of *set expressions* of sort $s$ is defined by

$$e_s := \{t_s, \ldots, t_s\} \mid s \mid (e_s \setminus e_s)$$

where $\{t_s, \ldots, t_s\}$ is possibly empty. The language $Props(X)$ of *propositions* is defined by

$$\phi := e_s \subseteq e_s \mid (\phi \wedge \phi) \mid (\neg\phi)$$

An *expression* is a term, a set expression, or a proposition. □

The concrete syntax to be used in this chapter and that will be implemented by the COMMUNITY Workbench [WNF99] is defined as follows.

**Notation 4.5.** An identifier is a sequence of letters, digits and underscores, starting with a letter.

*Id* := *Letter* (*Letter* | *Digit* | '_')*
*Letter* := 'a' | ... | 'z' | 'A' | ... | 'Z'
*Digit* := '0' | ... | '9'

The following reserved words may not be used as identifiers:

```
bool do false head in init int out prog skip tail true
```

□

**Notation 4.6.** There are only three pre-defined types: integers and booleans, with the usual constants and operations, and lists. Propositions are boolean expressions. A value for a list is written $[l_1, l_2, \ldots]$ and thus [] is the empty list. The operations 'head' and 'tail' perform as usual, and '+' represents list concatenation.

The following grammar shows the ASCII and mathematical versions of the operators. We use only the latter in this chapter.

*Expression* := *Disjunction* ('=>' | '⇒' *Disjunction*)*
*Disjunction* := *Conjunction* ('|' | '∨' *Conjunction*)*
*Conjunction* := *Relation* ('&' | '∧' *Relation*)*
*Relation* := *Addition* (*RelOp Addition*)*
*RelOp* := '=' | '/=' | '≠' | '<' | '⊂' | '>' | '⊃' | '<=' | '≤' | '⊆' | '>=' | '≥' | '⊇'
*Addition* := *Multiplication* ( ('+' | '-' | '∪' | '\') *Multiplication*)*
*Multiplication* := *Opposite* (('*' | '/' | 'mod' | '∩') *Opposite*)*
*Opposite* := ('-' | '~' | '¬')* *Primary*
*Primary* := *Id* | *Integer* | *Boolean* | *List* | *Set* | '(' *Expression* ')'
*Integer* := *Digit*+
*Boolean* := 'true' | 'false'
*List* := '[' [*Expression* (',' *Expression*)*] ']' | ('head' | 'tail') '(' *Expression* ')'
*Set* := *Type* | '{' [*Expression* (',' *Expression*)*] '}' | '∅'
*Type* := 'int' | 'bool' | 'list' '(' *Type* ')'

Booleans may be compared only with the (in)equality operators: expressions like `false < true` are syntactically wrong. All binary operators are left-associative. □

### 4.2.2 Semantics

Not only the syntax of expressions, but the semantics also must be independent of the chosen abstract data types. Therefore we assume a fixed algebra that satisfies the properties of the operators.

**Definition 4.7.** An *algebra* $\mathcal{A}$ assigns a set $s_{\mathcal{A}}$ to each sort $s \in S$, and a total function $f_{\mathcal{A}} : s_{1_{\mathcal{A}}} \times \cdots \times s_{n_{\mathcal{A}}} \to s_{\mathcal{A}}$ to each function symbol $f \in F_{\langle s_1,\ldots,s_n\rangle,s}$ such that the axioms $\Phi$ are satisfied. □

The semantics of an expression can thus be given in the usual way, provided values are assigned to the occurring variables.

**Definition 4.8.** A *valuation* for a set of variables $X$ is an $S$-indexed family of functions $\mathcal{V}_s : X_s \to s_{\mathcal{A}}$. □

**Definition 4.9.** Given a valuation $\mathcal{V}$ for variables $X$, the *denotation* of a term $t \in Terms(X)$, written $[\![t]\!]_{\mathcal{V}}$, is given as follows:

- $[\![v]\!]_{\mathcal{V}} = \mathcal{V}_s(v)$
- $[\![c]\!]_{\mathcal{V}} = c_{\mathcal{A}}$
- $[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{V}} = f_{\mathcal{A}}([\![t_1]\!]_{\mathcal{V}},\ldots,[\![t_n]\!]_{\mathcal{V}})$

for any $v \in X_s$, $c \in F_{\langle\rangle,s}$, $f \in F_{\langle s_1,\ldots,s_n\rangle,s}$ and $t_i \in Terms_{s_i}(X)$. The denotation $[\![e]\!]_{\mathcal{V}}$ of an expression $e \in Sets(X)$ is

- $[\![\{t_1,\ldots,t_n\}]\!]_{\mathcal{V}} = \{[\![t_1]\!]_{\mathcal{V}},\ldots,[\![t_n]\!]_{\mathcal{V}}\}$
- $[\![s]\!]_{\mathcal{V}} = s_{\mathcal{A}}$
- $[\![(e \setminus e')]\!]_{\mathcal{V}} = [\![e]\!]_{\mathcal{V}} \setminus [\![e']\!]_{\mathcal{V}}$

with $t_i \in Terms_s(X)$ and $e, e' \in Sets(X)$. A proposition $\phi \in Props(X)$ is true, written $\mathcal{V} \models \phi$, in the following cases:

- $\mathcal{V} \models e \subseteq e'$ if and only if $[\![e]\!]_{\mathcal{V}} \subseteq [\![e']\!]_{\mathcal{V}}$;
- $\mathcal{V} \models (\phi \wedge \phi')$ if and only if $\mathcal{V} \models \phi$ and $\mathcal{V} \models \phi'$;
- $\mathcal{V} \models (\neg\phi)$ if and only if $\mathcal{V} \not\models \phi$. □

The notion of validity is the usual one.

**Definition 4.10.** A proposition $\phi \in Props(X)$ is *valid*, written $\models_X \phi$, if $\mathcal{V} \models \phi$ for any valuation $\mathcal{V}$ for $X$. □

### 4.2.3 Configuration

Different sets of variables may be related by a sort-preserving mapping.

**Definition 4.11.** Let $X$ and $X'$ be two sets of variables. A *variable morphism* $\sigma : X \to X'$ is a function such that $\sigma(X_s) \subseteq X'_s$. □

A variable morphism $\sigma : X \to X'$ induces an obvious translation of the languages built over $X$ into those over $X'$.

**Notation 4.12.** Let $x$ be an expression over $X$. Then $\sigma(x)$ is the expression over $X'$ obtained by replacing each variable $v$ of $x$ by $\sigma(v)$. □

The following result is needed in the next section.

**Proposition 4.1.** *Sets of variables and their morphisms form a finitely cocomplete category.* □

*Proof.* It is immediate that the functional composition of variable morphisms is a variable morphism. The remaining properties come from $\mathfrak{Set}$. In particular, the pushout of $\{\sigma_i : X_0 \to X_i\}$ (with $i = 1, 2$) is the same as the union of the pushouts of $\{\sigma_i : X_{0_s} \to X_{i_s}\}$ for each sort $s$. ✓

In general, given a morphism and a "model" for its codomain, we wish to be able to construct a reduct, i.e., a model for its domain, thus providing the semantic basis of the morphism. The next definition applies this idea to variable morphisms.

**Definition 4.13.** Given a variable morphism $\sigma : X \to X'$ and a valuation $\mathcal{V}'$ for $X'$, the $\sigma$-*reduct* of $\mathcal{V}'$ is the valuation $\mathcal{V}$ for $X$ such that for any $v \in X$ one has $\mathcal{V}(v) = \mathcal{V}'(\sigma(v))$. □

Since the value of an expression depends only on the values of its variables, it is obvious that it is not changed by reducts.

**Lemma 4.2.** *Let $\mathcal{V}$ be the $\sigma$-reduct of $\mathcal{V}'$, with $\sigma : X \to X'$. Then $[\![e]\!]_{\mathcal{V}} = [\![\sigma(e)]\!]_{\mathcal{V}'}$ for any $e \in \mathrm{Terms}(X) \cup \mathrm{Sets}(X)$, and $\mathcal{V} \models \phi$ if and only if $\mathcal{V}' \models \sigma(\phi)$ for any $\phi \in \mathrm{Props}(X)$.* □

It results that morphisms do not affect the validity of formulas.

**Lemma 4.3.** *Given a variable morphism $\sigma : X \to X'$, if $\models_X \phi$ then $\models_{X'} \sigma(\phi)$.* □

## 4.3 Signatures

The "vocabulary" of a program is given by its signature. It defines, together with the abstract data types, what expressions can be written in the body of the program to control and perform computations. Moreover, the signature is visible to the environment and provides the means to establish interactions.

This section shows how signatures are specified, what is their meaning, and how the signature of the system can be obtained from the signatures of the components.

### 4.3.1 Syntax

A program signature is a set of variables and a set of action names. Each variable is either for input—its value is provided by the environment and cannot be changed by the program—or for output—its value is initialised by the program and modified only by its actions. In other words, input variables are read-only and thus not under the control of the program, while output variables can be read and written, and the program has full control over their values.

The signature also states for each action the set of output variables it modifies, called the action's domain. Inversely, the domain of an output variable is the set of actions that change it. Each program also includes a so called idle action which is always executable and which performs nothing, i.e., it does not change any of the program's output variables. It corresponds to a computation step performed by the environment and hence allows to see any computation as an infinite sequence of actions, as is usual for reactive languages [MP91].

**Definition 4.14.** A *program signature* is a tuple $\langle I, O, A \rangle$ where

- $I$ is a set of variables called *input variables*,
- $O$ is a set of variables called *output variables*,
- $A = \bigcup_{d \subseteq O} A_d$ is a set, whose elements are called *actions*, with a distinguished element $\bot \in A_\emptyset$ called the *idle action*.

The sets $O$, $I$ and $A_d$ are finite and mutually disjoint. The *domain* of an action $a \in A$ is the set $d \subseteq O$ such that $a \in A_d$. □

**Notation 4.15.** The program variables are $V = \bigcup_{s \in S} V_s = \bigcup_{s \in S}(O_s \cup I_s)$. The sort of variable $v$ is denoted by $sort(v)$. The domain of action $a$ is denoted by $D(a)$. Inversely, for each $o \in O$ the set of actions that can change $o$ is $D(o) = \{a \in A \mid o \in D(a)\}$. □

**Notation 4.16.** An action's domain can be considered its type, and therefore we write concrete signatures in the form

$$\langle \{i_1 : sort(i_1), \ldots\}, \{o_1 : sort(o_1), \ldots\}, \{a_1 : D(a_1), \ldots\} \rangle$$

□

### 4.3.2 Semantics

The semantics of a program is given by a labelled transition system where transitions are labelled by actions and states are labelled by valuations for the program's variables.

**Definition 4.17.** An *interpretation* for a program signature is a labelled transition system with $L_T = A$ and $L_W$ the collection of valuations for $V$. □

**Notation 4.18.** We write $\mathcal{V}_w$ for $lbl_W(w)$. □

The above definition allows many transition systems to be interpretations of the execution of some program. However, not all of them are meaningful. In particular, we are only interested in those that obey the domains of actions: if the action executing at some step does not include the output variable $o$ in its domain, then the value of $o$ remains the same.

**Definition 4.19.** A *model* for a program signature is an interpretation such that for any $o \in O$ and any $a \in L_T \setminus D(o)$, if $\mathcal{V}_w \xrightarrow{a} \mathcal{V}_{w'}$ then $\mathcal{V}_w(o) = \mathcal{V}_{w'}(o)$. □

### 4.3.3 Configuration

This section shows how the signature $\psi$ of the program $P$ representing the whole system can be obtained from the signatures $\psi_{1,\ldots,n}$ of the programs $P_{1,\ldots,n}$ representing the components.

Making use of the categorical framework, the system's signature will be the colimit of the diagram showing the configuration of the components' signatures. From the definition of colimit, there must be a morphism from each $\psi_i$ to $\psi$. The definition of a signature morphism $\sigma_i : \psi_i \to \psi$ must therefore capture the intuition that, $P_i$ being a component (or part or "sub-program") of $P$, $\psi_i$ is included in $\psi$. Put differently, $\psi$ is obtained from the addition of the variables and actions of the various $\psi_i$ and therefore signature morphisms capture a notion of superposition, $P_i$ being the underlying programs and $P$ the "transformed" one.

To show inclusion, superposition is achieved by a mapping from $P_i$'s vocabulary into that of $P$. In the initial definitions of COMMUNITY [FM97, FL97], a signature morphism mapped each variable and action of the component into *one* variable or action of the system such that the type and domain of variables and actions were preserved. For this work we have kept the basic intuition but introduced a small although fundamental change [WF98a].

In a reconfiguration setting, a program may synchronise each of its actions with different actions from different programs at different times. To allow this, a morphism may

associate an action $a$ of a component $P_i$ with a *set* of actions $\{a_1, \ldots, a_m\}$ of the system $P$. The intuition is that those actions correspond to the behaviour of $a$ when synchronising with other actions of the other components $P_{j \neq i}$ of $P$. The morphism is quite general: the set $\{a_1, \ldots, a_m\}$ may be empty. In that case, action $a$ has been effectively removed from $P$. This contradicts the intuition that every action of the component should also be in the system, but it is necessary to handle configurations with conflicting requirements (see Example 4.2 on page 58). A further constraint is necessary to be able to build a system signature from any configuration of component signatures (Proposition 4.6 on page 58): internal synchronisation is not possible. This means that two distinct actions of a component may not be mapped to the same action of the system. Mathematically, their image sets must be disjoint.

The definitions and proofs become easier if the mapping of actions is done by a function in the opposite way, i.e., if a mapping $a \mapsto \{a_1, \ldots, a_m\}$ becomes a set of mappings $a \leftarrow\!\shortmid a_1, \ldots, a \leftarrow\!\shortmid a_m$. However, in examples and informal discussions we use the "set version" of the action mapping.

We now state the complete requirements for a signature morphism to capture a superposition relation between the underlying signature of the component and the transformed signature of the system.

1. Each variable of the component is mapped to a variable of the system of the same type.
2. Component output variables are mapped to system output variables.
3. Each action of the system is mapped to an action of the component.
4. The idle action of the system is mapped to the idle action of the component.
5. The domain of variables is preserved, i.e., all system actions that modify a given variable must correspond to component actions that change the corresponding component variable, if it exists.
6. The domain of actions is preserved, i.e., if a component action modifies a given component output variable, then the corresponding system actions modify the corresponding system output variable.

The second requirement states that the variables under control of the component are also under control of the system. The inverse is not true: input variables of a component $P_i$ which are under the control of another component $P_j$ become output variables of any system $P$ containing $P_i$ and $P_j$. The fifth condition states that the domain of an output variable cannot be extended by system actions that are unrelated to the component actions that modify the variable. This enforces that a program's output variables are only under its control, even if the program is combined with other programs into a larger one. It corresponds to the requirement in UNITY that new actions may only modify the superposed variables, they cannot contain assignments to the underlying variables. The sixth requirement states that the domain of a component action is not restricted by the system, i.e., all corresponding system actions do not change less variables.

**Definition 4.20.** Given program signatures $\psi = \langle I, O, A \rangle$ and $\psi' = \langle I', O', A' \rangle$, a *signature morphism* $\sigma : \psi \to \psi'$ consists of a variable morphism $\sigma_v : V \to V'$ and a function $\sigma_a : A' \to A$ such that

1. $\sigma_v(O) \subseteq O'$
2. $\sigma_a(\bot') = \bot$
3. $\forall o \in O \; \sigma_a(D'(\sigma_v(o))) \subseteq D(o)$

4. $\forall a' \in A'\ \sigma_v(D(\sigma_a(a'))) \subseteq D'(a')$ □

Our first result is that signatures and their morphisms constitute a category. This basically asserts that morphisms can be composed. In other words, superposition is transitive (and reflexive, of course).

**Proposition 4.4.** *Program signatures and signature morphisms constitute a category* $\mathfrak{Sig}$. □

*Proof.* We only have to show that the composition $\sigma;\sigma' = \langle \sigma_v;\sigma'_v, \sigma'_a;\sigma_a\rangle$ is well-defined, i.e., it returns a signature morphism, because the remaining properties come from Proposition 4.1 on page 52 and $\mathfrak{Set}$. Let $\sigma : \psi_1 \to \psi_2$ and $\sigma' : \psi_2 \to \psi_3$ be two signature morphisms.

1. $\sigma_v(O_1) \subseteq O_2 \wedge \sigma'_v(O_2) \subseteq O_3 \Rightarrow \sigma'_v(\sigma_v(O_1)) \subseteq O_3$
2. $\sigma'_a(\bot_3) = \bot_2 \wedge \sigma_a(\bot_2) = \bot_1 \Rightarrow \sigma_a(\sigma'_a(\bot_3)) = \bot_1$
3. From $\sigma_v(o_1) \in O_2$ and $\forall o_2 \in O_2\ \sigma'_a(D_3(\sigma'_v(o_2))) \subseteq D_2(o_2)$ we obtain
$$\forall o_1 \in O_1\ \sigma_a(\sigma'_a(D_3(\sigma'_v(\sigma_v(o_1))))) \subseteq \sigma_a(D_2(\sigma_v(o_1))) \subseteq D_1(o_1)$$
4. From $\sigma'_a(a_3) \in A_2$ and $\forall a_2 \in A_2\ \sigma_v(D_1(\sigma_a(a_2))) \subseteq D_2(a_2)$ we obtain
$$\forall a_3 \in A_3\ \sigma'_v(\sigma_v(D_1(\sigma_a(\sigma'_a(a_3))))) \subseteq \sigma'_v(D_2(\sigma'_a(a_3))) \subseteq D_3(a_3)$$

✓

**Notation 4.21.** In the following, the indices $v$ and $a$ are omitted. □

Given an interpretation for the signature of a system $P$, morphisms $\sigma_i : P_i \to P$ allow us to obtain interpretations for its components $P_i$. Basically, the interpretations have the same worlds, and every transition $\mathcal{V}_w \xrightarrow{a} \mathcal{V}_{w'}$ in the interpretation for $P$ is transformed into a transition $\mathcal{V}^i_w \xrightarrow{\sigma_i(a)} \mathcal{V}^i_{w'}$ for each $P_i$, where for any world $w$, $\mathcal{V}^i_w$ is the $\sigma_i$-reduct of $\mathcal{V}_w$.

**Definition 4.22.** Given a signature morphism $\sigma : \psi \to \psi'$ and an interpretation $\mathcal{I}'$ for $\psi'$, the *$\sigma$-reduct* is the interpretation $\mathcal{I}$ for $\psi$ such that

1. $\mathcal{I}$ and $\mathcal{I}'$ have the same underlying graph,
2. for any world $w$, $\mathcal{V}_w$ is the $\sigma$-reduct of $\mathcal{V}'_w$,
3. for any transition $t$, $\mathrm{lbl}(t) = \sigma(\mathrm{lbl}'(t))$. □

Since signature morphisms preserve action domains, reducts preserve models. This shows that the "part-of" intuition given at the syntactic level by the morphism has a semantic basis, given by the reduct.

**Proposition 4.5.** *If $\sigma : \psi \to \psi'$ is a signature morphism then the $\sigma$-reduct of every model of $\psi'$ is a model of $\psi$.* □

*Proof.* We take as hypothesis the antecedent of the condition for being a model (Definition 4.19 on page 54):
$$\forall o \in O\ \forall a \in L_T \setminus D(o)\ \mathcal{V}_{w_1} \xrightarrow{a} \mathcal{V}_{w_2}$$
Due to item 1 of the definition of $\sigma$-reduct, $\mathcal{I}'$ must have a corresponding transition $\mathcal{V}'_{w_1} \xrightarrow{a'} \mathcal{V}'_{w_2}$ with $a = \sigma(a')$. We prove $a' \in L'_T \setminus D'(\sigma(o))$ by absurd. Assume $a' \in D'(\sigma(o))$.

Then $\sigma(a') \in \sigma(D'(\sigma(a)))$ and therefore $a \in D(o)$ contradicting our assumption $a \notin D(o)$. So we have

$$\forall o \in O\ \forall a \in L_T \setminus D(o)\ \exists a' \in L'_T \setminus D'(\sigma(o))\ \mathcal{V}'_{w_1} \xrightarrow{a'} \mathcal{V}'_{w_2}$$

By assumption, $\mathcal{I}'$ is a model and thus

$$\forall o \in O\ \forall a \in L_T \setminus D(o)\ \mathcal{V}'_{w_1}(\sigma(o)) = \mathcal{V}'_{w_2}(\sigma(o))$$

Due to item 2 in the $\sigma$-reduct definition, we can apply Definition 4.13 on page 53, obtaining as wished

$$\forall o \in O\ \forall a \in L_T \setminus D(o)\ \mathcal{V}_{w_1}(o) = \mathcal{V}_{w_2}(o)$$

✓

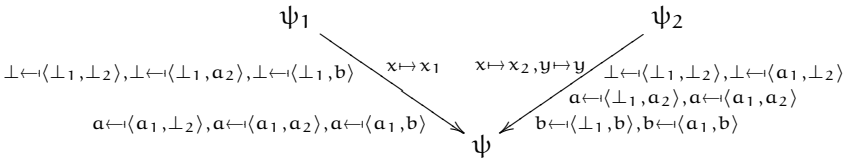
So far we have only considered the relationship between components and the system. We now turn to the main goal of this section: how to build the signature of the system given the signatures of the components. The basic idea is that the interactions between programs are specified by a diagram in $\mathfrak{Sig}$, and the colimit returns the signature of the system. In the following, we show how the superposition relation given by signature morphisms can be used to specify interactions and how the colimits indeed capture the restrictions on variables and actions imposed by the interactions.

In the simplest case, there are no interactions and thus the colimit returns the parallel composition of the components. In Category Theory, all relationships between objects must be made explicit through morphisms. In the particular case of CommUnity programs, it means for example that two variables (or actions) of two unrelated programs are different, even if they have the same name. Therefore the variables of the system are the disjoint union of the components' variables. Regarding actions, the parallel composition contains all possible combinations of actions from the components, since there is no restriction on their co-occurrence. Mathematically, the actions of the colimit are the cartesian product of the actions of the components, thus providing not an interleaving but a concurrent semantics for parallel composition.

*Example 4.1.* Consider the following two components without input variables, written using Notation 4.16 on page 54:

- $\psi_1 = \langle \emptyset, \{x : int\}, \{\bot : \emptyset, a : \{x\}\}\rangle$
- $\psi_2 = \langle \emptyset, \{x : int, y : int\}, \{\bot : \emptyset, a : \{x\}, b : \{y\}\}\rangle$

The colimit is given by the diagram




with $\psi = \langle \emptyset, \{x_1 : int, x_2 : int, y : int\}, \{\langle \bot_1, \bot_2\rangle : \emptyset, \langle \bot_1, a_2\rangle : \{x_2\}, \langle \bot_1, b\rangle : \{y\}, \langle a_1, \bot_2\rangle : \{x_1\}, \langle a_1, a_2\rangle : \{x_1, x_2\}, \langle a_1, b\rangle : \{x_1, y\}\}\rangle$ □

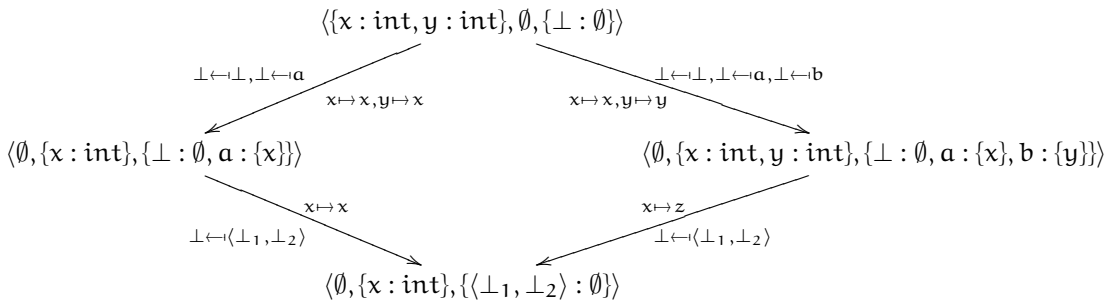
Interactions between programs are established through action synchronisation and memory sharing. This is achieved by relating the relevant action and variable names of the interacting programs through morphisms from a third, "mediating" program—called channel—that represents the common vocabulary. For example, to state that action $a_1$ of program $P_1$ is the same as (i.e., synchronises with) action $a_2$ of $P_2$, the channel $C$ contains just an action $a$ and two morphisms $\sigma_i : C \to P_i$ that map $a$ to $a_i$.

The colimit is then the parallel composition restricted to the sharing specified by the diagram. If an input variable of one component is shared with an output variable of another one, it becomes an output variable in the system. Concerning actions, we eliminate from the cartesian product all actions $\langle a_1, \ldots, a_i, \ldots, a_j, \ldots, a_n\rangle$ such that

1. $a_i$ and $a_j$ are not synchronised but $a_i$ is synchronised with (at least) one action of the j-th component,
2. $a_i$ modifies a variable that is shared with an output variable of the j-th component, but $a_j$ does not modify it.

The first condition enforces the synchronisations given by the diagram, the second requirement states that if two output variables are shared, the actions in their domains must be synchronised.

*Example 4.2.* Due to the second condition, sharing of variables can have side-effects on actions. Consider that the signatures of the previous example are to share all variables. We get the following colimit.



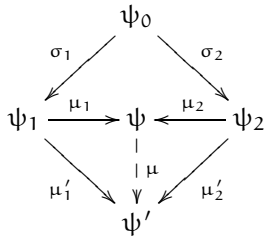

Notice that although no synchronisation is explicitly imposed on the actions, they all disappear. The intuitive reason is that the variables of $\psi_2$ become shared indirectly through the variable of $\psi_1$. To satisfy the constraints on the domains, the actions of $\psi_2$ must be synchronised, but internal synchronisation is not possible, and therefore they cannot be mapped to any action of the system.

Mathematically, it can be checked that, except for the idle action, no action of the colimit of the previous example satisfies the second condition above. For instance, $\langle \perp_1, a_2 \rangle$ is eliminated because $\perp_1$ does not modify $x_1$ which has become shared with the domain of $a_2$. Likewise for $\langle a_1, a_2 \rangle$, where $a_2$ does not change $y$ which is shared with the domain of $a_1$, the variable $x_1$. □

It remains to prove that any configuration of signatures has a semantics.

**Proposition 4.6.** *$\mathfrak{Sig}$ is finitely cocomplete.* □

*Proof.* The initial object is $\langle \emptyset, \emptyset, \{\perp\} \rangle$. As for pushouts, consider the diagram




The proof that $\{\mu_i : \psi_i \to \psi\}$ is a pushout of $\{\sigma_i : \psi_0 \to \psi_i\}$ has the following steps:

1. Define $\psi$ and $\mu_i$ and show their correctness.
2. Show that $\sigma_1; \mu_1 = \sigma_2; \mu_2$.
3. For any other pushout candidate $\{\mu'_i : \psi_i \to \psi'\}$, show there is a unique $\mu : \psi \to \psi'$ such that $\mu_i; \mu = \mu'_i$.

For variables, we take $\{\mu_i : V_i \to V\}$ the pushout over $\mathfrak{Set}$ (Proposition 4.1 on page 52), and define $O = \bigcup_{i=1,2} \mu_i(O_i)$ and thus $\mu_i(O_i) \subseteq O$ as required for signature morphisms. It remains to show that $\mu(O') \subseteq O$ because the existence and uniqueness of $\mu$ are guaranteed by $\mathfrak{Set}$:

$$\begin{array}{lcll} \mu(O) & = & \mu(\bigcup_{i=1,2} \mu_i(O_i)) & \text{definition of } O \\ & = & \bigcup_{i=1,2} \mu(\mu_i(O_i)) & \\ & = & \bigcup_{i=1,2} \mu'_i(O_i)) & \mu_i \text{ form a pushout} \\ & \subseteq & O' & \mu'_i \text{ are signature morphisms} \end{array}$$

We proceed now with the proof for actions.

1. We first define the set of shared output variables as $SO = \{\langle o_1, o_2\rangle \in O_1 \times O_2 \mid \mu_1(o_1) = \mu_2(o_2)\}$. The elements of $\psi$ are then:

   - $A = \{\langle a_1, a_2\rangle \in A_1 \times A_2 \mid \sigma_1(a_1) = \sigma_2(a_2) \wedge \forall \langle o_1, o_2\rangle \in SO\ o_1 \in D_1(a_1) \iff o_2 \in D_2(a_2)\}$
   - $\bot = \langle \bot_1, \bot_2\rangle$
   - $D(\langle a_1, a_2\rangle) = \bigcup_{i=1,2} \mu_i(D_i(a_i))$
   - $\mu_i = \pi_i$

   Now we show that $\mu_i$ are indeed signature morphisms.

   - $\mu_i(\bot) = \bot_i$ by definition
   - We prove the preservation of variable domains only for $i = 1$, the proof for $i = 2$ being similar.

$$\begin{array}{rcl} \mu_1(D(\mu_1(o_1))) & \subseteq & D_1(o_1) \\ a_1 \in \mu_1(D(\mu_1(o_1))) & \Rightarrow & a_1 \in D_1(o_1) \\ \langle a_1, a_2\rangle \in D(\mu_1(o_1)) & \Rightarrow & a_1 \in D_1(o_1) \\ \mu_1(o_1) \in D(\langle a_1, a_2\rangle) & \Rightarrow & o_1 \in D_1(a_1) \\ \mu_1(o_1) \in \bigcup_{i=1,2} \mu_i(D_i(a_i)) & \Rightarrow & o_1 \in D_1(a_1) \end{array}$$

   case 1:

$$\begin{array}{rl} & \mu_1(o_1) \in \mu_1(D_1(a_1)) \\ \Rightarrow & \exists o'_1 \in D_1(a_1)\ \mu_1(o_1) = \mu_1(o'_1) \\ \Rightarrow & \exists o'_1 \in D_1(a_1)\ \exists o_2 \in O_2\ \mu_1(o_1) = \mu_1(o'_1) = \mu_2(o_2) \\ \Rightarrow & \exists o'_1 \in D_1(a_1)\ \exists o_2 \in O_2\ \langle o'_1, o_2\rangle \in SO \wedge \langle o_1, o_2\rangle \in SO \\ \Rightarrow & \exists o'_1 \in D_1(a_1)\ \exists o_2 \in O_2\ o_2 \in D_2(a_2) \wedge \langle o_1, o_2\rangle \in SO \\ \Rightarrow & o_1 \in D_1(a_1) \end{array}$$

   case 2:

$$\begin{array}{rl} & \mu_1(o_1) \in \mu_2(D_2(a_2)) \\ \Rightarrow & \exists o_2 \in D_2(a_2)\ \mu_1(o_1) = \mu_2(o_2) \\ \Rightarrow & \exists o_2 \in D_2(a_2)\ \langle o_1, o_2\rangle \in SO \\ \Rightarrow & o_1 \in D_1(a_1) \end{array}$$

   - The preservation of action domains is also only proved for $\mu_1$:
     $\mu_1(D_1(\mu_1(\langle a_1, a_2\rangle))) = \mu_1(D_1(a_1)) \subseteq \bigcup_{i=1,2} \mu_i(D_i(a_i)) = D(\langle a_1, a_2\rangle)$

2. The commutativity of the diagram stems from the definition of $A$: $\sigma_1(\mu_1(\langle a_1, a_2\rangle)) = \sigma_1(a_1) = \sigma_2(a_2) = \sigma_2(\mu_2(\langle a_1, a_2\rangle))$

3. The uniqueness of $\mu$ is due to $\mu_i$ being projections: $\mu_1(\mu(a')) = \mu'_1(a') \wedge \mu_2(\mu(a')) = \mu'_2(a') \Rightarrow \mu(a') = \langle \mu'_1(a'), \mu'_2(a')\rangle$

   To prove the existence of $\mu$ it is necessary to show that for any $a' \in A'$, $\mu(a') = \langle \mu'_1(a'), \mu'_2(a')\rangle \in A$:

   (a) $\sigma_1(\mu'_1(a')) = \sigma_2(\mu'_2(a'))$ by definition of $\mu'_i$

   (b) We prove $\forall \langle o_1, o_2\rangle \in SO\ o_1 \in D_1(a_1) \iff o_2 \in D_2(a_2)\}$ only in one direction, the other being similar.

$$\begin{aligned} o_1 \in D_1(\mu'_1(a')) &\Rightarrow \mu'_1(o_1) \in \mu'_1(D_1(\mu'_1(a'))) \subseteq D'(a') \\ &\Rightarrow \mu(\mu_1(o_1)) = \mu(\mu_2(o_2)) = \mu'_2(o_2) \in D'(a') \\ &\Rightarrow a' \in D'(\mu'_2(o_2)) \\ &\Rightarrow \mu'_2(a') \in \mu'_2(D'(\mu'_2(o_2))) \subseteq D_2(o_2) \\ &\Rightarrow o_2 \in D_2(\mu'_2(a)) \end{aligned}$$

   It remains to show that $\mu$ is a signature morphism.

   - $\mu(\bot') = \langle \mu'_1(\bot'), \mu'_2(\bot')\rangle = \langle \bot_1, \bot_2\rangle = \bot$
   - Let $o \in O$. We know $\exists i \in \{1,2\}\ \exists o_i \in O_i\ \mu_i(o_i) = o$ and therefore assume a fixed $o_1$ such that $\mu_1(o_1) = o$.

$$\begin{aligned} \mu'_1(D'(\mu'_1(o_1))) &\subseteq D_1(o_1) \\ o_1 &\in D_1(\mu'_1(D'(\mu'_1(o_1)))) \\ \mu_1(o_1) &\in \mu_1(D_1(\mu'_1(D'(\mu'_1(o_1))))) \\ \mu_1(o_1) &\in \mu_1(D_1(\mu'_1(D'(\mu'_1(o_1))))) \cup \mu_2(D_2(\mu'_2(D'(\mu'_1(o_1))))) \\ \mu_1(o_1) &\in D(\langle \mu'_1(D'(\mu'_1(o_1))), \mu'_2(D'(\mu'_1(o_1)))\rangle) \\ \langle \mu'_1(D'(\mu'_1(o_1))), \mu'_2(D'(\mu'_1(o_1)))\rangle &\in D(\mu_1(o_1)) \\ \{\langle \mu'_1(D'(\mu'_1(o_1))), \mu'_2(D'(\mu'_1(o_1)))\rangle\} &\subseteq D(\mu_1(o_1)) \\ \mu(D'(\mu'_1(o_1))) &\subseteq D(\mu_1(o_1)) \\ \mu(D'(\mu(\mu_1(o_1)))) &\subseteq D(\mu_1(o_1)) \\ \mu(D'(\mu(o))) &\subseteq D(o) \end{aligned}$$

   - For the preservation of action domains we apply directly the definition of $D$.

$$\begin{aligned} \mu(D(\mu(a'))) &= \mu(D(\langle \mu'_1(a'), \mu'_2(a')\rangle)) \\ &= \mu(\mu_1(D_1(\mu'_1(a'))) \cup \mu_2(D_2(\mu'_2(a')))) \\ &= \mu'_1(D_1(\mu'_1(a'))) \cup \mu'_2(D_2(\mu'_2(a'))) \\ &\subseteq D'(a') \cup D'(a') \end{aligned}$$

✓

## 4.4 Programs

A program defines the initial values of its output variables and also when and how the actions modify them.

### 4.4.1 Syntax

An action is a guarded set (not a sequence!) of non-deterministic assignments, one for each variable in the action's domain. The right hand side of an assignment is an expression that denotes a set of values. This is useful to support underspecification [LF99]. At each step, one of the actions is selected and, if its guard is true, its assignments are executed simultaneously. This is done by first evaluating all the expressions and then, for each set obtained, assigning non-deterministically one of the values to the attribute on the left-hand side. The guard of an idle action is always true. Actions only state how output variables are modified. Their initial values are provided non-deterministically by a constraint.

**Definition 4.23.** A *program* $\langle \psi, \beta \rangle$ is a program signature $\psi$ with a *program body* $\beta = \langle ic, E, G \rangle$ where

1. $ic \in \mathrm{Props}(O)$ is the *initialisation condition*,
2. $E : A \times O \rightharpoonup \mathrm{Sets}(V)$ assigns to every $a \in A$ and to every $o \in D(a)$ a set expression of sort $\mathrm{sort}(o)$ and is undefined in all other cases;
3. $G : A \to \mathrm{Props}(V)$ assigns a *guard* to every action such that $\models_V G(\bot)$. □

**Notation 4.24.** The concrete syntax for programs is given by the following grammar.

| | |
|---|---|
| *Program* | := 'prog' *Id* [*InVars*] [*OutVars*] [*Init*] [*Actions*] |
| *InVars* | := 'in' *Vars* |
| *OutVars* | := 'out' *Vars* |
| *Vars* | := *VarList* (';' *VarList*)* |
| *VarList* | := *Id* (',' *Id*)* ':' *Type* |
| *Init* | := 'init' *Expression* |
| *Actions* | := 'do' *Action* ('[]' *Action*)* |
| *Action* | := *Id* ':' [*Expression* '->' \| '→'] *ActionBody* |
| *ActionBody* | := 'skip' \| *Assignment* ('‖' *Assignment*)* |
| *Assignment* | := *Id* (':=' \| ':∈') *Expression* |

When the initialisation condition or a guard is omitted, it is assumed to be true. We abbreviate $v :\in \{t\}$ as $v := t$. The command `skip` shows that the action has empty domain. We omit the idle action. □

*Example 4.3.* The program that controls a cart is

```
prog Cart
in     idest, ibag : int
out    loc, odest, obag : int
init   0 ≤ loc ≤ U-1 ∧ odest = -1 ∧ obag = 0
do     move: loc ≠ odest → loc := loc +U 1
[]     get: odest = -1 → obag := ibag ‖ odest := idest
[]     put: loc = odest → obag := 0 ‖ odest := -1
```

Bags are represented by integers, the absence of a bag being denoted by zero. Locations are represented by integers from zero to the track length minus one. Initially, the destination of the cart is an impossible location so that the cart keeps moving until it gets a bag and a valid gate location through action 'get'. When it reaches its destination, the cart unloads the bag through action 'put'. Notice that since input variables may be changed arbitrarily by the environment, the cart must copy their values to output variables to make sure the correct bag is unloaded at the correct gate. □

*Example 4.4.* To be able to compute how many bags are processed per lap on average, we add two counters initialised to zero. We memorise the current position so that we know when a lap has been completed. The bag counter is incremented when a bag is fetched from the check-in.

```
prog Cart_Stat
in    idest, ibag : int
out   loc, odest, obag, sloc, laps, bags : int
init  0 ≤ loc ≤ U-1 ∧ odest = -1 ∧ obag = 0 ∧ sloc = loc ∧ laps = 0 ∧ bags = 0
do    move: loc ≠ odest ∧ loc +U 1 ≠ sloc → loc := loc +U 1
[]    lap: loc ≠ odest ∧ loc +U 1 = sloc → loc := loc +U 1 ‖ laps := laps + 1
[]    get: odest = -1 → obag := ibag ‖ odest := idest ‖ bags := bags + 1
[]    put: loc = odest → obag := 0 ‖ odest := -1
```

□

*Example 4.5.* A check-in counter starts with a non-empty queue of bags, and loads one by one onto passing carts.

```
prog Check_In
out   loc, bag, dest : int; next : bool; q : list(int)
init  0 ≤ loc ≤ U-1 ∧ q ≠ [] ∧ next
do    new: q ≠ [] ∧ next → bag := head(q) ‖ q := tail(q) ‖ next := false
[]    put: ¬next → next := true
```

Variable 'next' is used to impose sequentiality among the actions. To build a system for our example, the 'put' action must be synchronised with a cart's 'get' action and variables 'bag' and 'dest' must be shared with 'ibag' and 'idest', respectively. □

*Example 4.6.* A gate starts with an empty queue of bags and adds each new bag to the front.

```
prog Gate
in    bag : int
out   loc : int; q : list(int)
init  0 ≤ loc ≤ U-1 ∧ q = []
do    get:  q := [bag] + q
```

In an architecture for our example, action 'get' must be synchronised with a cart's 'put' action, and variable 'bag' must be shared with 'obag'. □

As stated in the previous section, a channel contains the features that are shared between the programs it is linked to, thus establishing a symmetrical and partial relationship between the vocabularies of those programs. To be more precise, a channel is just a degenerate program that provides the basic interaction mechanisms (synchronisation and memory sharing) between given programs and thus adds no variables or computations of its own.

**Definition 4.25.** A *channel* is a program with true initialisation, no local variables, and all actions have true guards. □

**Notation 4.26.** A channel is simply abbreviated as $\langle I \mid A \rangle$ because it is always of the form

```
prog P
in    I
init  true
do      []   a: true → skip
      a∈A
```

□

### 4.4.2 Semantics

A model of a program makes precise the intuitive semantics outlined at the start of the previous section:

- the initial world satisfies the initialisation condition;
- a transition through action $a$ can occur in world $w$ if the action's guard is true in $w$;
- if the assignment $o :\in E(a, o)$ is executed in some world $w$, the value of $o$ in the next world is some element of the set obtained by evaluating $E(a, o)$ in $w$.

The last condition ensures that assignments are executed simultaneously (since all right-hand sides are evaluated in the same world) and that they are non-deterministic (if $E(a, o)$ contains at least two elements).

**Definition 4.27.** A *model* for a program is a model for its signature such that

1. $\mathcal{V}_{w_0} \models ic$,
2. for any $a \in A$, if $\mathcal{V}_w \xrightarrow{a} \mathcal{V}_{w'}$ then $\mathcal{V}_w \models G(a)$,
3. for any $o \in O$ and any $a \in D(o)$, if $\mathcal{V}_w \xrightarrow{a} \mathcal{V}_{w'}$ then $\mathcal{V}_{w'}(o) \in [\![E(a, o)]\!]_{\mathcal{V}_w}$. □

Notice that according to this definition an action is never executed if it includes an assignment $o :\in \emptyset$.

### 4.4.3 Configuration

Consider two programs $P$ and $P'$ such that there is a morphism from $P$'s signature to the one of $P'$. For the superposition relation to be kept, the body of $P'$ may not change in any way the actions of $P$ nor the initialisation condition for $P$'s variables. It may only add variables and actions (or assignments to existing actions). However, we wish to allow the user to refine programs too, by restricting the guards, the sets on right-hand sides of assignments, and the constraints on initial values. Summing up, $P$ is refined by $P'$ if $P$'s vocabulary is included in the one of $P'$ and if $P'$ does not relax the behaviour imposed by the initialisation condition and actions of $P$, i.e., if

- the initialization condition is not weakened,
- the guards are not weakened,
- the assignments are not less deterministic.

**Definition 4.28.** A *program morphism* $\sigma : \langle \psi, \beta \rangle \to \langle \psi', \beta' \rangle$ is a signature morphism $\sigma : \psi \to \psi'$ such that

1. $\models_{V'} ic' \Rightarrow \sigma(ic)$,
2. for any $a' \in A'$, $\models_{V'} G'(a') \Rightarrow \sigma(G(\sigma(a')))$,
3. for any $a' \in A'$ and any $o \in D(\sigma(a'))$, $\models_{V'} E'(a', \sigma(o)) \subseteq \sigma(E(\sigma(a'), o))$. □

**Notation 4.29.** We use the following conventions to avoid cluttering a diagram with all the morphisms' mappings.

- Action mappings are given in the same direction as variable mappings (and as the morphism), using set notation when necessary:

```
prog P                  x↦{y,z}   prog Q
do   x:  skip        ------------> do   y:  skip   means y↦x, z↦x
                                   []   z:  skip
```

- If an action is the result of synchronising other actions, its name is the concatenation of the names of those actions. Thus, by default, an action a is mapped to all actions that contain the name a; and a variable is mapped to one with the same name:

```
prog P                          prog Q
in   i : int        ------>     out   i, j : int
do   a:  skip                   do    ab:  skip   means i↦i, a↦{ab, ac}
                                []    ac:  skip
```

- The mappings are omitted when they can be unambiguously determined:

```
prog P                          prog Q
in    i : int                   in    x : bool
out   j : int; b : bool ------> out   y : bool; k : int   means i↦k, j↦k, b↦y
```

□

*Example 4.7.* It is obvious that the program of Example 4.3 on page 61 is a sub-program of the one in Example 4.4 on page 62, i.e., that morphism

$$\text{Cart} \xrightarrow{\text{move}\mapsto\{\text{move,lap}\}} \text{Cart\_Stat}$$

obeys the conditions of Definition 4.28 on the preceding page. □

The category of signatures extends to programs.

**Proposition 4.7.** *Programs and their morphisms constitute a category* $\mathfrak{Prog}$. □

*Proof.* Let $\sigma_1 : \psi_1 \to \psi_2$ and $\sigma_2 : \psi_2 \to \psi_3$ be two signature morphisms. It is obvious that the identity for signature morphisms is also an identity for program morphisms. It remains to show that composition is well-defined.

1. The steps of the proof are as follows.

| | | |
|---|---|---|
| (a) | $\models_{V_2} ic_2 \Rightarrow \sigma_1(ic_1)$ | definition of $\sigma_1$ |
| (b) | $\models_{V_3} \sigma_2(ic_2 \Rightarrow \sigma_1(ic_1))$ | Lemma 4.3 on page 53 |
| (c) | $\models_{V_3} \sigma_2(ic_2) \Rightarrow \sigma_2(\sigma_1(ic_1)))$ | Notation 4.12 on page 52 |
| (d) | $\models_{V_3} ic_3 \Rightarrow \sigma_2(ic_2)$ | definition of $\sigma_2$ |
| (e) | $\models_{V_3} ic_3 \Rightarrow \sigma_2(\sigma_1(ic_1))$ | from (c) and (d) |

2. Using the same technique we get:

| | | |
|---|---|---|
| (a) | $\models_{V_3} E_3(a_3, \sigma_2(\sigma_1(o_1))) \subseteq \sigma_2(E_2(\sigma_2(a_3), \sigma_1(o_1)))$ | $\sigma_1(o_1) \in O_2$ |
| (b) | $\models_{V_2} E_2(\sigma_2(a_3), \sigma_1(o_1)) \subseteq \sigma_1(E_1(\sigma_1(\sigma_2(a_3)), o_1))$ | $\sigma_2(a_3) \in A_2$ |
| (c) | $\models_{V_3} E_3(a_3, \sigma_2(\sigma_1(o_1))) \subseteq \sigma_2(\sigma_1(E_1(\sigma_1(\sigma_2(a_3)), o_1)))$ | |

3. (a) $\models_{V_3} G_3(a_3) \Rightarrow \sigma_2(G_2(\sigma_2(a_3)))$ $a_3 \in A_3$
   (b) $\models_{V_2} G_2(\sigma_2(a_3)) \Rightarrow \sigma_1(G_1(\sigma_1(\sigma_2(a_3))))$ $\sigma_2(a_3) \in A_2$
   (c) $\models_{V_3} \sigma_2(G_2(\sigma_2(a_3))) \Rightarrow \sigma_2(\sigma_1(G_1(\sigma_1(\sigma_2(a_3)))))$
   (d) $\models_{V_3} G_3(a_3) \Rightarrow \sigma_2(\sigma_1(G_1(\sigma_1(\sigma_2(a_3)))))$ ✓

The next result shows that the definition of program morphism indeed captures a notion of refinement.

**Proposition 4.8.** *If $\sigma : P \to P'$ is a program morphism then the reduct of every model of $P'$ is a model of $P$.* □

*Proof.* Assume $\mathcal{I}'$ is a model for $P'$ and let $\mathcal{I}$ be the $\sigma$-reduct of $\mathcal{I}'$.

1. (a) $\mathcal{V}'_{w_0} \models ic'$ assumption
   (b) $\mathcal{V}'_{w_0} \models ic' \Rightarrow \sigma(ic)$ Definition 4.28 on page 63 and Definition 4.10 on page 52
   (c) $\mathcal{V}'_{w_0} \models \sigma(ic)$
   (d) $\mathcal{V}_{w_0} \models ic$ Lemma 4.2 on page 53

2. Using Definition 4.22 on page 56 we have

   (a) $\mathcal{V}_{w_1} \xrightarrow{a} \mathcal{V}_{w_2}$ hypothesis
   (b) $\mathcal{V}'_{w_1} \xrightarrow{a'} \mathcal{V}'_{w_2}$ with $a = \sigma(a')$ and $\mathcal{V}_{w_i}$ a $\sigma$-reduct of $\mathcal{V}'_{w_i}$
   (c) $\mathcal{V}'_{w_1} \models G'(a')$ assumption
   (d) $\mathcal{V}'_{w_1} \models \sigma(G(\sigma(a')))$ $\sigma$ definition
   (e) $\mathcal{V}_{w_1} \models G(a)$ Lemma 4.2 on page 53

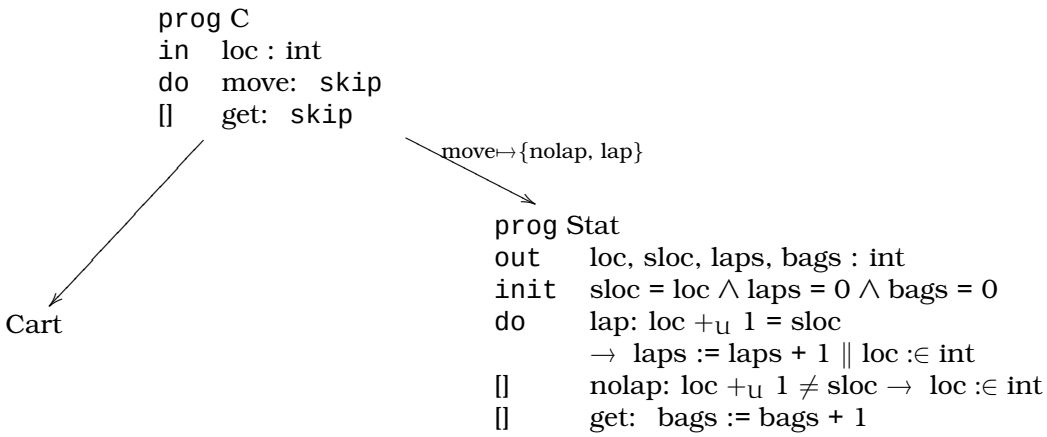
3. Let $o \in O$ and $a \in D(o)$.

   (a) $\mathcal{V}_{w_1} \xrightarrow{a} \mathcal{V}_{w_2}$ hypothesis
   (b) $\mathcal{V}'_{w_1} \xrightarrow{a'} \mathcal{V}'_{w_2}$ as above
   (c) $\mathcal{V}'_{w_2}(\sigma(o)) \in [\![E'(a', \sigma(o))]\!]_{\mathcal{V}'_{w_1}}$ $a' \in D(\sigma(o))$ and assumption
   (d) $\mathcal{V}'_{w_2}(\sigma(o)) \in [\![\sigma(E(\sigma(a'), o))]\!]_{\mathcal{V}'_{w_1}}$ $\sigma$ definition and Definition 4.9 on page 52
   (e) $\mathcal{V}_{w_2}(o) \in [\![E(a, o)]\!]_{\mathcal{V}_{w_1}}$ Definition 4.13 on page 53 and Lemma 4.2 on page 53

   The justification for the third step is the following chain of implications using Definition 4.20 on page 55: $a \in D(o) \Rightarrow \sigma(a') \in D(o) \Rightarrow o \in D(\sigma(a')) \Rightarrow \sigma(o) \in \sigma(D(\sigma(a'))) \Rightarrow \sigma(o) \in D'(a') \Rightarrow a' \in D'(\sigma(o))$. ✓

The colimit of a diagram of programs is easy to compute once the colimit over signatures is calculated: the initialisation condition is the conjunction of all the initialisation conditions in the diagram, and for each action $\langle a_1, \ldots, a_n \rangle$ of the colimit, its guard is the conjunction of the guards of all $a_i$ and its assignments are the union of the assignments of all $a_i$. If several $a_i$ assign to a shared variable, the right-hand side is the intersection of the respective right-hand sides.

*Example 4.8.* The colimit of




is the Cart_Stat program of Example 4.4 on page 62. □

**Proposition 4.9.** *Category $\mathfrak{Prog}$ is finitely cocomplete.* □

*Proof.* The pushout of $\sigma_i : \langle \psi_0, \beta_0 \rangle \to \langle \psi_i, \beta_i \rangle$ with $i = 1, 2$ is $\{\mu_i : \langle \psi_i, \beta_i \rangle \to \langle \psi, \beta \rangle\}$ with $\{\mu_i : \psi_i \to \psi\}$ the pushout over signatures as given in the proof of Proposition 4.6 on page 58 and $\beta$ defined by

1. $ic = \bigwedge_{i=1,2} \mu_i(ic_i)$
2. for any $a \in A$, $G(a) = \bigwedge_{i=1,2} \mu_i(G_i(\mu_i(a)))$
3. for any $a \in A$, $\forall o \in D(a)\ E(a, o) = \bigcap \{\mu_i(E_i(\mu_i(a), o_i)) \mid i \in \{1, 2\} \wedge \mu_i(o_i) = o\}$

For the last definition to be meaningful, we must have $o \in D(a) \Rightarrow o_i \in D_i(\mu_i(a))$ for $i = 1, 2$. Due to the characterisation of $o$ and signature pushout, it is equivalent to $\mu_i(o_i) \in \bigcup_{j=1,2} \mu_j(D_j(a_j)) \Rightarrow o_i \in D_i(a_i)$ with $a = \langle a_1, a_2 \rangle$. This has been shown in the proof of Proposition 4.6 on page 58.

To prove it is a pushout it is enough to prove that $\mu_1$, $\mu_2$, and $\mu$ are program morphisms, if $\mu_1'$ and $\mu_2'$ are. From the definition of $\beta$ it is immediate that $\mu_1$ and $\mu_2$ satisfy the conditions of Definition 4.28 on page 63. As for $\mu$ the proofs are as follows:

1. (a) $\models_{V'} ic' \Rightarrow \mu_1'(ic_1)$ and $\models_{V'} ic' \Rightarrow \mu_1'(ic_1)$ — definition of $\mu_i'$
   (b) $\models_{V'} ic' \Rightarrow \bigwedge_{i=1,2} \mu_i'(ic_i)$
   (c) $\models_{V'} ic' \Rightarrow \bigwedge_{i=1,2} \mu(\mu_i(ic_i))$ — $\mu_i$ are pushout of variables
   (d) $\models_{V'} ic' \Rightarrow \mu(\bigwedge_{i=1,2} \mu_i(ic_i))$ — $\mu$ is a substitution
   (e) $\models_{V'} ic' \Rightarrow \mu(ic)$ — definition of $ic$
2. Using the same reasoning one has the following steps.
   (a) $\models_{V'} G'(a') \Rightarrow \bigwedge_{i=1,2} \mu_i'(G_i(\mu_i'(a')))$
   (b) $\models_{V'} G'(a') \Rightarrow \bigwedge_{i=1,2} \mu(\mu_i(G_i(\mu_i(\mu(a')))))$
   (c) $\models_{V'} G'(a') \Rightarrow \mu(\bigwedge_{i=1,2} \mu_i(G_i(\mu_i(\mu(a')))))$
   (d) $\models_{V'} G'(a') \Rightarrow \mu(G(\mu(a')))$
3. Let $o \in D(\mu(a'))$.

(a) $E'(a', \mu(o)) = \{E'(a', \mu(\mu_i(o_i))) \mid i \in \{1,2\} \wedge \mu_i(o_i) = o\}$
(b) $E'(a', \mu(o)) = \{E'(a', \mu'_i(o_i)) \mid i \in \{1,2\} \wedge \mu_i(o_i) = o\}$
(c) $E'(a', \mu(o)) = \bigcap\{E'(a', \mu'_i(o_i)) \mid i \in \{1,2\} \wedge \mu_i(o_i) = o\}$
(d) $E'(a', \mu(o)) \subseteq \bigcap\{\mu'_i(E_i(\mu'_i(a'), o_i)) \mid i \in \{1,2\} \wedge \mu_i(o_i) = o\}$
(e) $E'(a', \mu(o)) \subseteq \bigcap\{\mu(\mu_i(E_i(\mu_i(\mu(a')), o_i))) \mid i \in \{1,2\} \wedge \mu_i(o_i) = o\}$
(f) $E'(a', \mu(o)) \subseteq \mu(\bigcap\{\mu_i(E_i(\mu_i(\mu(a')), o_i)) \mid i \in \{1,2\} \wedge \mu_i(o_i) = o\})$
(g) $E'(a', \mu(o)) \subseteq \mu(E(\mu(a'), o))$

The definition of program morphism can be applied in the fourth step because each possible $o_i$ is in the condition of item 3 in Definition 4.28 on page 63:

(a) $o \in D(\mu(a'))$
(b) $\mu_i(o_i) \in \bigcup_{j=1,2} \mu_j(D_j(\mu_j(\mu(a'))))$ definition of D
(c) $o_i \in D_i(\mu_i(\mu(a')))$ see proof of Proposition 4.6 on page 58
(d) $o_i \in D_i(\mu'_i(a'))$ ✓

As we have seen in Example 4.2 on page 58, sharing of output variables might have unexpected side-effects on the actions that are available. These problems can only happen if there is no morphism between the programs sharing the variable because superposition preserves the domain of a variable (i.e., the set of actions that manipulate it). We want to rule out those diagrams. This restriction forces interactions between programs to be synchronous *communication* of values (from output variables to input variables), a very general mode of interaction that is suitable for the modular development of reusable components, as needed for architectural design.

We take advantage of the graphical nature of diagrams to present a simple and intuitive definition that has a straightforward efficient implementation. The fundamental idea is trivial: if there is an undirected path (through the morphisms) between two output variables, they are shared. If there is no directed path between them, there is no superposition relation between the corresponding programs and therefore the condition is violated.

We begin by extracting from the diagram of programs the relevant information about sharing of variables. Since we work only with directed graphs, to forget the direction of arrows in a path, each mapping between two variables must generate a pair of opposite arcs between them.

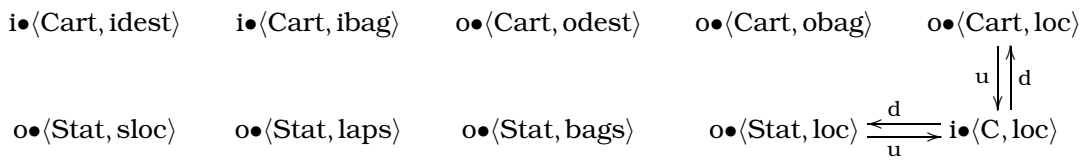
**Definition 4.30.** The *data view* for a diagram in $\mathfrak{Prog}$ is a graph labelled over $L_N = \{i, o\}$ and $L_A = \{d, u\}$ with one node for each variable of each program in the diagram, and one arc labelled "d" from node $n$ to $n'$ if and only if there is a morphism in the diagram mapping the variable corresponding to $n$ to the one that corresponds to $n'$. For each arc labelled "d" there is exactly one arc labelled "u" in the opposite direction. A node is labelled "i" if and only if it corresponds to an input variable. □

**Notation 4.31.** A node in the data view is written $o\bullet\langle P_j, o_{jk}\rangle$ or $i\bullet\langle P_j, i_{jk}\rangle$ to show both the label as well as the variable to which it corresponds. Since the label can be determined from the kind of the variable, we omit it in some examples. Inversely, when the variables corresponding to the nodes are not relevant, we only show the labels. □

Now we check only the paths between output variables. Their arcs must be equally labelled. If the label is "d" then there is a morphism from the source of the path to the target. If the label is "u" the morphism is in the opposite direction.

**Definition 4.32.** A diagram in $\mathfrak{Prog}$ is *well-formed* if in its data view whenever there is a path between a pair of distinct nodes labelled "o", there is a path with all arcs with the same label. □

*Example 4.9.* The diagram in Example 4.8 on page 65 is not well-formed. Its data view is




This graph does not satisfy the required condition: there is a path

$$o\bullet \xrightarrow{u} i\bullet \xrightarrow{d} o\bullet$$

but no

$$o\bullet \xrightarrow{d} i\bullet \xrightarrow{d} o\bullet$$

or

$$o\bullet \xrightarrow{u} i\bullet \xrightarrow{u} o\bullet\ .$$

□

As we see, the condition is quite strong, as it rules out diagrams that can be meaningful. However, we should emphasize again that the rationale is for each value to be generated only in one place. Since values are generated by computations on output variables, it means that each output variable and its domain (i.e., its associated "methods") are defined in one program only, which may be then "inherited" by others.

We further restrict the kind of diagrams we are interested in, since architectures are obtained in a very particular way: through the application of connectors (Section 4.6 on page 72) and through the replacement of components by more specialized ones (Section 4.8 on page 96).

First we observe that, although finite cocompleteness allows any program to establish interactions between other programs, for the diagram to be well-formed in general the mediating program may not contain output variables, because otherwise the data view would contain the subgraph

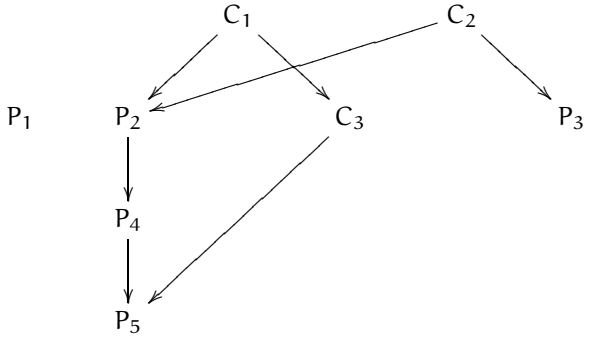
$$o\bullet \underset{u}{\overset{d}{\rightleftarrows}} o\bullet \underset{u}{\overset{d}{\rightleftarrows}} o\bullet\ .$$

We therefore require that interactions are only specified by programs especially designed for that purpose, namely channels (Definition 4.25 on page 62).

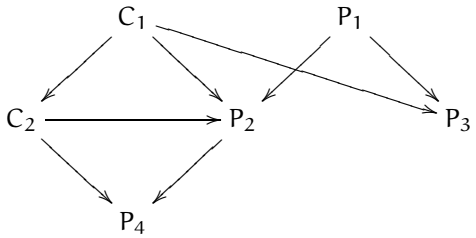
The configurations we consider are then well-formed diagrams where each program is either isolated, or connected through $n$ channels to exactly $n$ other programs (i.e., channels establish interactions pairwise), or a specialisation of one or more programs.

**Definition 4.33.** A *configuration* is a well-formed diagram in $\mathfrak{Prog}$ where each node either has outdegree less than two or else is labelled with a channel and has indegree zero and outdegree two. □

*Example 4.10.* Let $C_i$ be channels and $P_i$ programs that are not channels. Then



is a configuration if it is well-formed, but




is not a configuration because $C_1$ has outdegree 3, $P_1$ is not a channel, and $C_2$ has indegree 1. □

It is efficient to check whether a given diagram is a configuration. Notice that, due to the restriction on the outdegree of programs, in the data view each output variable has at most one outgoing arc, and thus it is easy to see whether there is a directed path between two of them. Moreover, the find-union algorithm [CLR90] can compute efficiently the connectivity of variables, since the actual non-directed paths are irrelevant.

## 4.5 Program Instances

A program instance provides a snapshot of the state of a program.

### 4.5.1 Syntax

A program instance is defined as a program together with a function that assigns to each output variable a term (over some fixed set of so called logical variables) of the same sort. Two explanations are in order.

First, the function may return an arbitrary term, not just a ground term. Although in the running system the value of each program variable is given by a ground term, we need logical variables to be able to write reconfiguration rules whose left-hand sides match against programs with possibly infinite distinct combinations of values for their variables.

The second point worth noticing is that terms are not assigned to input variables. This contrasts with our first approach [FWM99] and there are several reasons. The pragmatic one is that an input variable is often an output variable in some other component, and therefore there is no need to duplicate the specification of its value. The conceptual reason is that in this way we only represent the state that is under direct control of the component. The absence of the input variables' values makes clear that they may change at any moment. There is also a technical reason. Reconfigurations change the connectors between components. This entails that an input variable may become shared with a different output variable from a different component. If input variables would have explicitly represented state, the reconfiguration rule would have to change their state. However, that is not possible within the algebraic graph rewrite framework (see Section 4.8 on page 96).

**Definition 4.34.** Given a fixed set $LV$ of variables, called *logical variables*, a *program instance* $\langle P, \epsilon \rangle$ is a program $P$ with an *environment* $\epsilon : O \to Terms(LV)$ such that $\epsilon(O_s) \subseteq Terms_s(LV)$. □

A channel instance is the same as a channel. Therefore we only use the term "channel".

*Example 4.11.* The logical variables needed for our examples are $LV = \{l_n, b_n, i_n : \text{int}; r_n : \text{bool}; q_n : \text{list(int)}\}$ with $n \in \mathbb{N}$. We write $x_0$ simply as $x$. □

**Notation 4.35.** We represent program instances in tabular form:

| P | |
|---|---|
| $o_1$ | $\epsilon(o_1)$ |
| $o_2$ | $\epsilon(o_2)$ |
| $\vdots$ | $\vdots$ |

If P has no output variables, $\epsilon$ is empty and we write simply $\boxed{P}$. □

*Example 4.12.* The program instance

| Cart_Stat | |
|---|---|
| loc | $l$ |
| odest | $-1$ |
| obag | $\emptyset$ |
| sloc | $l +_U 1$ |
| laps | $i + 1$ |
| bags | $b$ |

represents a cart that has completed at least one lap and will complete another one with the next move. □

We extend $\epsilon$ to any expression using only output variables, and introduce a function that returns the variables actually used by $\epsilon$.

**Notation 4.36.** If $e$ is an expression over O, then $\epsilon(e) \in Terms(LV)$ is obtained by replacing in $e$ each $o$ by $\epsilon(o)$. We write $Vars(\langle P, \epsilon\rangle)$ for the minimal subset of LV such that for any $o \in O$, $\epsilon(o) \in Terms(Vars(\langle P, \epsilon\rangle))$. □

### 4.5.2 Semantics

A program instance represents all states (reachable from the initial state) of all models of the program that can match the terms given by the instance.

**Definition 4.37.** A *model* for a program instance $\langle P, \epsilon\rangle$ is a triple $\langle \mathcal{I}, w, \mathcal{V}\rangle$ where $w$ is a world of the model $\mathcal{I}$ for P and $\mathcal{V}$ is a valuation for LV, such that there is a path in $\mathcal{I}$ from $w_0$ to $w$ and for any $o \in O$ one has $[\![\epsilon(o)]\!]_{\mathcal{V}} = \mathcal{V}_w(o)$. □

### 4.5.3 Configuration

A morphism between program instances is a program morphism that preserves state.

**Definition 4.38.** A *program instance morphism* $\sigma : \langle P, \epsilon\rangle \to \langle P', \epsilon'\rangle$ is a program morphism $\sigma : P \to P'$ such that for any $o \in O$ one has $\models_{LV} \epsilon(o) = \epsilon'(\sigma(o))$. □

*Example 4.13.* Using the program instance of Example 4.12, the Example 4.7 on page 64 may be extended to

| Cart | |
|---|---|
| loc | $l$ |
| odest | $-1$ |
| obag | $l - l$ |

$\xrightarrow{\text{move}\mapsto\{\text{move,lap}\}}$

| Cart_Stat | |
|---|---|
| loc | $l$ |
| odest | $-1$ |
| obag | $\emptyset$ |
| sloc | $l +_U 1$ |
| laps | $i + 1$ |
| bags | $b$ |

□

Again, a morphism induces a reduction on the models.

**Proposition 4.10.** *Given a program instance morphism* $\sigma : \langle P, \epsilon \rangle \to \langle P', \epsilon' \rangle$ *and a model* $\langle \mathcal{I}', w', \mathcal{V}' \rangle$ *for* $\langle P', \epsilon' \rangle$*, the* $\sigma$-reduct $\langle \mathcal{I}, w', \mathcal{V}' \rangle$*, where* $\mathcal{I}$ *is the* $\sigma$*-reduct of* $\mathcal{I}'$*, is a model of* $\langle P, \epsilon \rangle$*.* □

*Proof.* Since $\mathcal{I}$ and $\mathcal{I}'$ have the same underlying graph, we only have to prove that $[\![\epsilon(o)]\!]_{\mathcal{V}'} = \mathcal{V}_{w'}(o)$ holds for any $o \in O$. Using the definitions of program instance morphism, validity, and $\sigma$-reduct of valuations we have $[\![\epsilon(o)]\!]_{\mathcal{V}'} = [\![\epsilon'(\sigma(o))]\!]_{\mathcal{V}'} = \mathcal{V}'_{w'}(\sigma(o)) = \mathcal{V}_{w'}(o)$. ✓

As expected,

**Proposition 4.11.** *Program instances and their morphisms form a category* $\mathfrak{Inst}$*.* □

*Proof.* Morphisms have identity and are compositional because equality of terms is reflexive and transitive, respectively. ✓

**Notation 4.39.** Given a diagram $D$ in $\mathfrak{Inst}$, we write $Vars(D)$ for the union of $Vars(\langle P, \epsilon \rangle)$ taken for each program instance in $D$. □

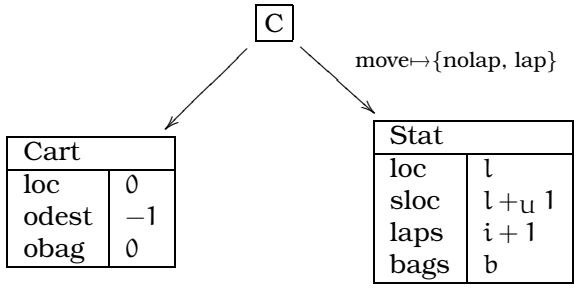
Every diagram in $\mathfrak{Inst}$ can be transformed into a diagram in the category of programs and program morphisms just by omitting the environment function.

**Proposition 4.12.** $\mathcal{IP} : \mathfrak{Inst} \to \mathfrak{Prog}$ *with* $\mathcal{IP}_N(\langle P, \epsilon \rangle) = P$ *and* $\mathcal{IP}_A$ *the identity map is a functor.* □

*Proof.* Immediate from Definition A.15 on page 111 with $\mathcal{F}_A(f) = f$. ✓

Contrary to the previous categories, $\mathfrak{Inst}$ is not finitely cocomplete.

*Example 4.14.* Applying $\mathcal{IP}$ to the following diagram we obtain the one in Example 4.8 on page 65.




Because of the 'Cart' instance, the pushout would have to assign zero to 'loc', but the 'Stat' instance does not impose any specific value to the location and the instance morphism to the pushout would have to preserve those possibilities. Summing up, $\not\models_{LV} 0 = l$ and therefore this diagram has no pushout. □

We thus need to extend the notion of well-formedness.

**Definition 4.40.** A diagram $D$ in $\mathfrak{Inst}$ is well-formed if $\mathcal{IP}(D)$ is. □

In a well-formed diagram output variables are not shared, only "inherited". Hence two different output variables have no conflicting state. Therefore any well-formed program instance diagram has a colimit, given by the colimit of the underlying program diagram together with the union of the environments given by the instances.

**Proposition 4.13.** *Every finite well-formed* $\mathfrak{Inst}$*-diagram has colimit.* □

*Proof.* Let D be such a diagram. If $\{\mu_i : P_i \to P\}$ is the colimit of $\mathcal{IP}(D)$ then $\{\mu_i : \langle P_i, \epsilon_i\rangle \to \langle P, \epsilon\rangle\}$ is the colimit of D with $\epsilon(o) = \epsilon_i(o_i)$ for some $\mu_i(o_i) = o$.

Assume there is a $P_j \neq P_i$ such that $\mu_j(o_j) = o$. Due to the property of colimits in $\mathfrak{Set}$, there is a program $P_k$ and a variable $v_k \in V_k$ such that $\sigma_{ki}(v_k) = o_i$ and $\sigma_{kj}(v_k) = o_j$ (i.e., $o_i$ and $o_j$ are shared through $v_k$). Thus in the data view we have

$$o\bullet\langle P_j, o_j\rangle \underset{d}{\overset{u}{\rightleftarrows}} \langle P_k, v_k\rangle \underset{u}{\overset{d}{\rightleftarrows}} o\bullet\langle P_i, o_i\rangle .$$

Due to well-formedness, there is a path

$$o\bullet\langle P_i, o_i\rangle \xrightarrow{d^*} o\bullet\langle P_j, o_j\rangle$$

(or in the opposite direction). Thus there is by composition a morphism $\sigma_{ij} : P_i \to P_j$ with $\sigma_{ij}(o_i) = o_j$. So in D we have $\epsilon_i(o_i) = \epsilon_j(\sigma_{ij}(o_i)) = \epsilon_j(o_j)$ and thus $\epsilon(o)$ does not depend on the chosen $i$.

We have proven that the $\mu_i$ are program instance morphisms and hence the colimit is well-defined. To prove that it is a colimit indeed, consider any other colimit candidate $\{\mu'_i : \langle P_i, \epsilon_i\rangle \to \langle P', \epsilon'\rangle\}$. It suffices to prove that the unique program morphism $\mu : P \to P'$ is also a program instance morphism. Consider any $o \in O$. Then there is a $o_i$ such that $\mu_i(o_i) = o$ and we have $\epsilon(o) = \epsilon(\mu_i(o_i)) = \epsilon_i(o_i) = \epsilon'_i(\mu'_i(o_i)) = \epsilon'(\mu(\mu_i(o_i))) = \epsilon'(\mu(o))$. ✓

It remains to extend the notion of configuration in the obvious way.

**Definition 4.41.** A *configuration instance* is a diagram D in $\mathfrak{Inst}$ such that $\mathcal{IP}(D)$ is a configuration. □

## 4.6 Connectors

Channels only allow us to express simple and static interactions between programs, namely synchronous communication of values, through sharing of input with output variables and sharing of actions. To express more complex or transient interactions, we use connectors. The next subsection provides the basic definitions, Section 4.6.2 on page 75 provides a list of connectors to be used in our example, and finally we show how the connectors can capture temporary interactions in a systematic way and how the categorical framework provides a way to define new connectors from existing ones.

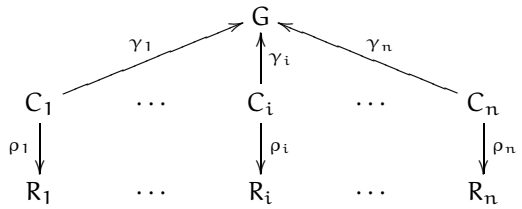
### 4.6.1 Definitions

A connector consists of a glue and one ore more roles. The roles constrain what components the connector can be applied to—thus acting like "formal parameters" of the connector—and the glue provides the interaction—thus being the "body" of the connector. There is no superposition relation between a role and the glue or vice-versa. Some elements of the roles (e.g., variables, actions, etc., depending on the language used to describe roles) only restrict what components can be matched by the role and are not needed for the interaction, while the glue introduces new elements not present in the roles to specify the coordination between the computations of the interacting programs. To sum up, we need mediators to state which elements of the roles are relevant for the glue.

In a categorical framework, the connectors that can be built depend on the categories used to represent glues, roles, and the mediators, and on the relationships between those categories. It is possible to use three different categories for the three parts of a connector (e.g., [FL97] proposes roles to be specifications written in temporal logic,

mediators to be program signatures, and glues to be programs). However, to relate the roles via the mediators to the glue it is necessary to give the connector as a diagram, which is necessarily in a single category. Therefore, there must exist functors which translate the three categories into a common one. For our work, we consider that category to be $\mathfrak{Prog}$ or $\mathfrak{Inst}$ and thus assume the original specification of the connector to have been already translated into one of them. We therefore adopt and adapt only the basic definitions of [FL97].

**Definition 4.42.** An $n$-*ary connector* (with $n > 0$) is a configuration of the form




Programs $G$ and $R_i$ are called *glue* and *roles*, respectively. Morphisms $\gamma_i$ and $\rho_i$ are called *glue* and *role morphisms*, respectively. A *connector instance* is a diagram $D$ in $\mathfrak{Inst}$ such that $\mathfrak{IP}(D)$ is a connector. □

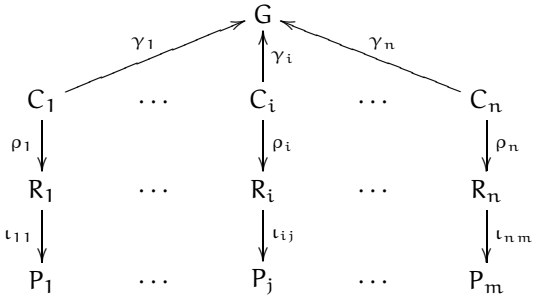
This definition, together with Definition 4.33 on page 68 and Definition 4.40 on page 71, entails

1. each program $C_i$ is a channel;
2. a connector (instance) is well-formed.

The first property just reflects our expectations as to how mediators are represented in CommUnity, making apparent that mediators are different from the usual roles and glues. The second one prevents a connector (instance) from introducing any problems concerning the sharing of variables when it becomes part of a larger diagram representing the architecture.

The categorical framework not only makes the relationships between the three parts of a connector explicit—each channel and pair of morphisms $\langle \gamma_i, \rho_i \rangle$ states which part of the vocabulary of the $i$-th role is used by the glue for the interaction—it also allows one to make precise when and how an $n$-ary connector can be applied to components $P_1, \ldots, P_m$, namely when morphisms $\iota_{ij} : R_i \to P_j$ exist. This corresponds to the intuition that the "actual arguments"—the components—must instantiate the "formal parameters"—the roles. Morphisms $\iota_{ij}$ thus capture refinement, while morphisms $\gamma_i$ and $\rho_i$ denote superposition. It is possible to have two different morphism definitions to distinguish them [Lop99], but in the end there must be a diagram with a single kind of morphism on which colimits can be computed. For that purpose we use again diagrams in $\mathfrak{Prog}$ or $\mathfrak{Inst}$.

**Definition 4.43.** An *applied* $n$-*ary connector (instance)* is a configuration (instance) of the form

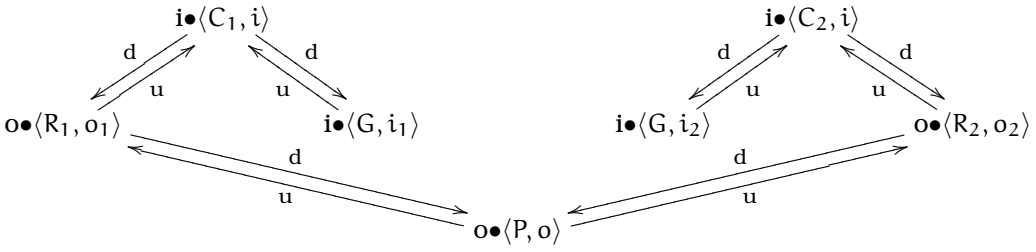


with $0 < m \leq n$. □

Two different roles may be refined by the same program and therefore we can have $m < n$.

The definition requires explicitly the diagram to be a configuration for two reasons. First, a role cannot be applied to more than one program. Second, without the requirement, the diagram might not be well-formed, although the connector by definition is. We present two simple examples of such cases.

*Example 4.15.* The following two data views correspond to invalid applications of connectors because they lead to shared output variables. The first achieves that by instantiating both roles with the same program.




The second obtains the same effect through superposition of an input variable of the role on an output variable of the component. The problem arises from the fact that the input variable corresponds to an output variable of the glue.

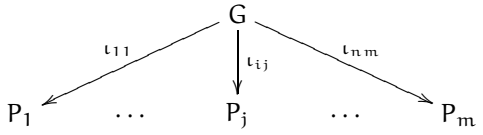
$$o\bullet\langle P, o\rangle \underset{u}{\overset{d}{\rightleftarrows}} i\bullet\langle R, i\rangle \underset{u}{\overset{d}{\rightleftarrows}} i\bullet\langle C, i\rangle \underset{u}{\overset{d}{\rightleftarrows}} o\bullet\langle G, o\rangle$$

□

**Notation 4.44.** We use $\{G \xleftarrow{\gamma_i} C_i \xrightarrow{\rho_i} R_i\}_n$ to denote an $n$-ary connector and $\{G \xleftarrow{\gamma_i} C_i \xrightarrow{\rho_i} R_i \xrightarrow{\iota_{ij}} P_j\}_{n,m}$ for its application. We omit $n$ and $m$ when they are irrelevant. □

If we take a black-box view of connector application it becomes apparent that a connector is an extension of a channel for more complex interactions that require additional computations and conditions, provided by the glue's actions through their assignments and guards. In fact, from the point of view of the user who is only interested in applying pre-defined connectors, not in changing their source code, a connector looks like an $n$-ary channel.

**Notation 4.45.** We name connectors after their glues. The *black-box view* of an applied connector is a labelled graph obtained from the diagram of the applied connector where the connector is collapsed into a single node labelled with its name, i.e., the name of the glue:




□

We emphasize that the black-box view is not a diagram in the categorical sense, but simply a graph that provides a convenient short-hand notation for configurations of multiple connectors and components.

The semantics of an (applied) connector (instance) is given by the colimit of the respective diagram. Moreover, the semantics of the applied version is a specialisation of

the semantics of the unapplied version. This means that the system obtained through application of connectors to components indeed enforces the interactions established by the connectors.

**Proposition 4.14.** *Given an applied connector* $D = \{G \xleftarrow{\gamma_i} C_i \xrightarrow{\rho_i} R_i \xrightarrow{\iota_{ij}} P_j\}$, *there is a program morphism from the colimit of* $\{G \xleftarrow{\gamma_i} C_i \xrightarrow{\rho_i} R_i\}$ *to the colimit of* $D$. □

*Proof.* The colimit of $\{G \xleftarrow{\gamma_i} C_i \xrightarrow{\rho_i} R_i\}$ is a colimit of $D' = \{G \xleftarrow{\gamma_i} C_i \xrightarrow{\rho_i} R_i \xrightarrow{id} R_i\}$ and $D$ specialises $D'$. The statement is then a consequence of Proposition A.6 on page 115. ✓

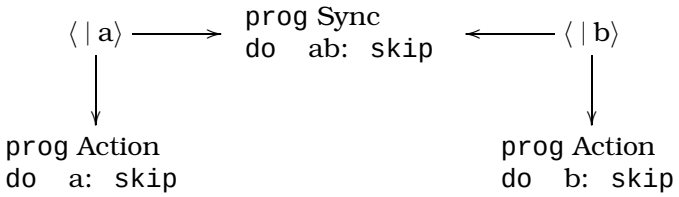
### 4.6.2 Catalog

We now present the connectors necessary for the remaining of this chapter and take the opportunity to show how some of the interactions proposed by MOBILE UNITY [RMP97, MR98] can be cast in our framework.

#### Synchronisation

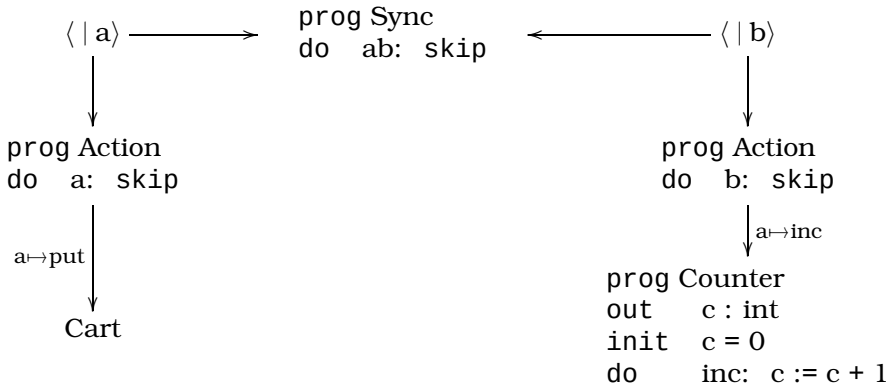
We begin with the connector that allows us to synchronise two actions of different programs. A channel would suffice for this purpose, but it is not able to capture the general case of transient synchronisation. Having already a connector for the simpler case makes the presentation more uniform.

**Definition 4.46.** The *synchronisation connector* is




□

*Example 4.16.* If we wish to count how often a cart unloads a bag, we monitor its 'put' action with a counter, i.e., we synchronise 'put' with an action that is always enabled and hence does not restrict the occurrence of the monitored action.




The colimit of this applied connector is

```
prog CountingCart
in    idest, ibag : int
out   loc, odest, obag, c : int
init  0 ≤ loc ≤ U-1 ∧ odest = -1 ∧ obag = 0 ∧ c = 0
do    move: loc ≠ odest → loc := loc +U 1
[]    get: odest = -1 → obag := ibag ‖ odest := idest
[]    put: loc = odest → obag := 0 ‖ odest := -1 ‖ c := c + 1
```

□

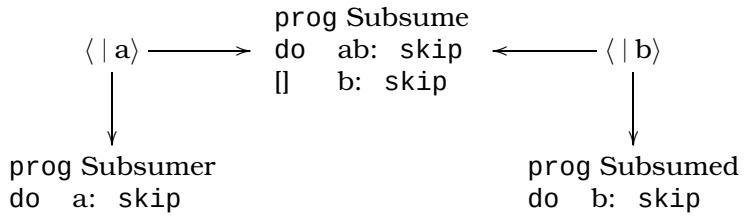
Due to the colimit semantics of a connector, for the synchronised action to occur, both guards of the original actions must be true. Hence, when two actions synchronise either both execute simultaneously or none is executed.

This contrasts with the approach taken by MOBILE UNITY which allows two kinds of synchronisation: coexecution and coselection [MR96]. The former corresponds to the notion exposed above, while the latter forces the two actions to be selected simultaneously but if one of them is inhibited or its guard is false then only the other action executes. This extends the basic semantics of UNITY where only one action can be selected at a time. Therefore, we do not handle coselection. Moreover, our intuitive notion of synchronisation corresponds to coexecution.

**Subsumption**

The logical analogy to synchronisation is equivalence. However, to avoid a cart $c_1$ colliding with the cart $c_2$ right in front of it we only need implication: if $c_1$ moves, so must $c_2$, but the opposite is not necessary. The analogy with implication also extends to the counter-positive: if $c_2$ cannot move, e.g., because it is (un)loading a bag, then neither can $c_1$. We call this "one-way" synchronisation action subsumption. For our example, the movement of $c_1$ subsumes the movement of $c_2$. As for the connector, it simply adds to the synchronisation connector the ability to let the subsumed action occur freely. This is only possible because our signature morphisms allow an action to ramify into a set of actions. In this case, the movement action of $c_2$ ramifies in two: one for the case in which it must co-occur with the movement of $c_1$, the other when it can occur freely.

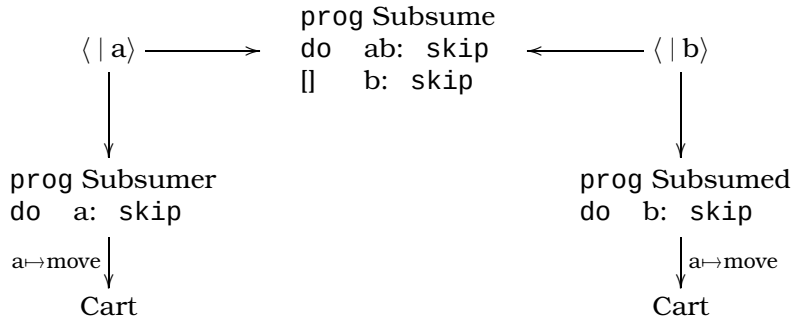
**Definition 4.47.** The *action subsumption* connector is




□

Notice that although the two roles are isomorphic, the binary connector is not symmetric because the glue treats the two actions differently. This is clearly indicated in the glue: 'b' may be executed alone at any time, while 'a' must co-occur with 'b'. Hence, action 'a' is the one that we want to connect to the 'move' action of $c_1$, while action 'b' is associated to the movement of $c_2$, as shown next.

*Example 4.17.* The applied connector is




with colimit

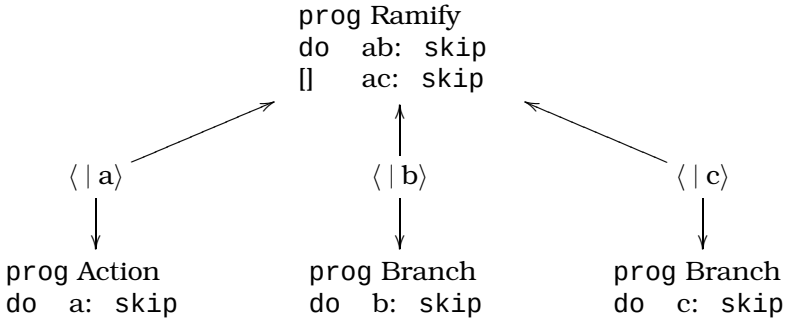
```
prog CollisionFreeCarts
in      idest1, ibag1, idest2, ibag2 : int
out     loc1, odest1, obag1, loc2, odest2, obag2 : int
init    ⋀_{i=1,2} 0 ≤ loci ≤ U-1 ∧ odesti = -1 ∧ obagi = 0
do      move1move2: loc1 ≠ odest1 ∧ loc2 ≠ odest2
        → loc1 := loc1 +_U 1 ‖ loc2 := loc2 +_U 1
[]      move2: loc2 ≠ odest2 → loc2 := loc2 +_U 1
[]      get1: odest1 = -1 → obag1 := ibag1 ‖ odest1 := idest1
[]      put1: loc1 = odest1 → obag1 := 0 ‖ odest1 := -1
[]      get2: odest2 = -1 → obag2 := ibag2 ‖ odest2 := idest2
[]      put2: loc2 = odest2 → obag2 := 0 ‖ odest2 := -1
[]      get1get2: odest1 = -1 ∧ odest2 = -1
        → obag1 := ibag1 ‖ odest1 := idest1 ‖ obag2 := ibag2 ‖ odest2 := idest2
[]      get1put2: odest1 = -1 ∧ loc2 = odest2
        → obag1 := ibag1 ‖ odest1 := idest1 ‖ obag2 := 0 ‖ odest2 := -1
[]      put1get2: loc1 = odest1 ∧ odest2 = -1
        → obag1 := 0 ‖ odest1 := -1 ‖ obag2 := ibag2 ‖ odest2 := idest2
[]      put1put2: loc1 = odest1 ∧ loc2 = odest2
        → obag1 := 0 ‖ odest1 := -1 ‖ obag2 := 0 ‖ odest2 := -1
```

□

**Ramification**

A generalisation of the previous connectors is to allow an action to synchronise with two actions of two different programs. The importance of this connector becomes apparent in Section 4.6.3 on page 85, where it is shown to be a primitive from which other connectors can be built.

**Definition 4.48.** The *action ramification* connector is



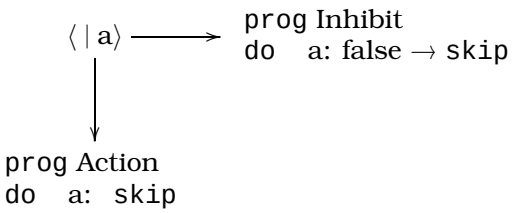

□

### Inhibition

In MOBILE UNITY, inhibition is a language primitive. In our framework it is simply a particular connector.

To inhibit an action we must let its guard become false. Due to the semantics of colimit, this can be done without changing the guard directly. It suffices to synchronise the action with one that has a false guard.

**Definition 4.49.** The *inhibition* connector is




□

### Asynchronous Communication

In MOBILE UNITY communication is achieved through variable sharing. The interaction $x \approx y$ `when` C `engage` I `disengage` $F_x \parallel F_y$ states the sharing condition C, the shared initial value I of both variables, and the final value $F_x$ and $F_y$ of each variable. The operational semantics states that whenever a program changes $x$, $y$ gets the same value, and vice-versa. As we have seen in Section 4.4.3 on page 63, this approach violates the locality principle that a variable under the control of one program should not be changed by another. Furthermore, as pointed out in [MR96], several restrictions have to be imposed in order to avoid problems like, e.g., simultaneous assignments of different values to shared variables.

We also feel that communication is a more appropriate concept than sharing for the setting we are considering, namely mobile agents that engage into transient interactions. In the framework of COMMUNITY programs, communication can be seen as some kind of sharing of local and input variables, which keeps the locality principle. We say "some kind" because we cannot use the same mechanism as in the static case, in which sharing means to map two different variables of the components into a single one of the system obtained by the colimit. In the mobile case the same output variable may be shared with different input variables at different times, and vice-versa. If we were to apply the usual construction, all those variables would become a single one in the resulting system, which is clearly unintended. Technically, the diagram would not be well-formed.

Therefore we obtain the same effect as transient sharing using a communication perspective. To be more precise, we assume a sender wants to transmit a message M, which is a set of output variables. If a receiver wants to get the message, it must provide input variables $M'$ which correspond in number and type to those of M. The sender produces the values, stores them in M, and waits for the message to be read by the receiver. Since COMMUNITY programs are not sequential, "waiting" has to be understood in a restricted sense. We only assume the sender does not produce another message before the previous one has been read (i.e., messages are not lost); it may however be executing other unrelated actions. To put it in another way, after producing M, the sender is expecting an acknowledge to produce the new values for the variables in M. For that purpose, we assume sender has an action 'put' which must be executed before the new message is produced. Similarly, program receiver must be informed when a new message has arrived, so that it may start processing it. For that purpose we

assume that a receiver has a single action 'get' which is the first action to be executed upon the receipt of a new message. That action may simply use $M'$ directly or it may copy it to output variables of the receiver.

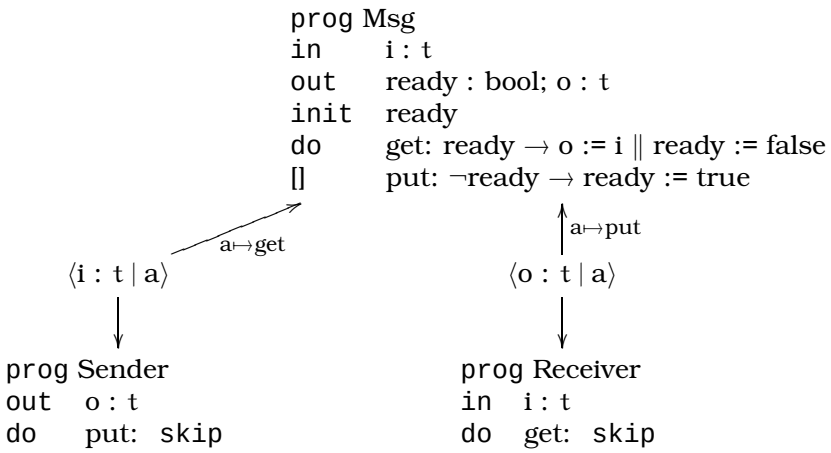
To sum up, communication is established via one single action for each program, similar in spirit to pointed processes in the $\pi$-calculus, or to ports in distributed systems: the action 'put' of sender is blocking (in the restricted sense mentioned above) until $M$ is transmitted, and similarly action 'get' of the receiver is inhibited while no new values have been transferred to $M'$. As expected, it is up for the glue of the interaction connector to transfer the values from $M$ to $M'$ and to impose this asynchronous communication policy.

The solution is to explicitly model the message transmission as the parallel assignment of the message variables, which we abbreviate as $M' := M$. For this to be possible, the output variables $M$ of the sender must be input variables of the glue, and the input variables $M'$ of the receiver must be output variables of the glue. The glue's actions are also symmetrical to those of the sender and receiver: there is a 'get' action to be synchronised with the action 'put' of the sender, thus performing the transmission in parallel with the notification of the sender, and there is a 'put' action to be synchronised with the 'get' action of the receiver.

To make sure that the connector forwards the message to the receiver only after getting it from the sender we use the same technique as in Example 4.5 on page 62, a boolean variable to impose sequentiality. This decouples the sender's action from the receiver's, thus imposing the asynchronicity.

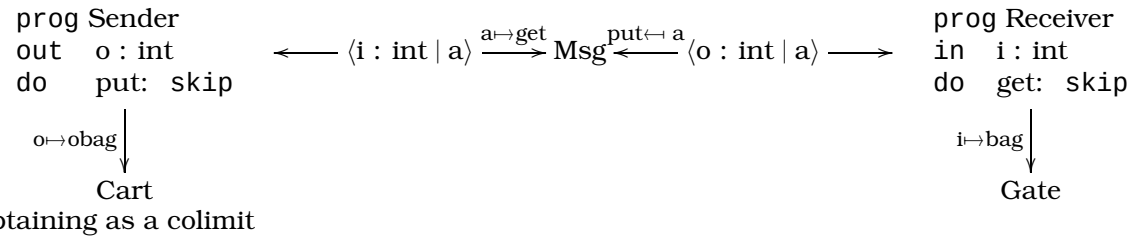
The connector presented next only transmits a single variable. It can be trivially generalised to messages as a set of variables (see also Example 4.26 on page 87).

**Definition 4.50.** The *asynchronous communication*, or *message passing*, connector to transmit a single variable of sort t is




□

*Example 4.18.* To unload a bag (t = int) we apply the above connector as follows



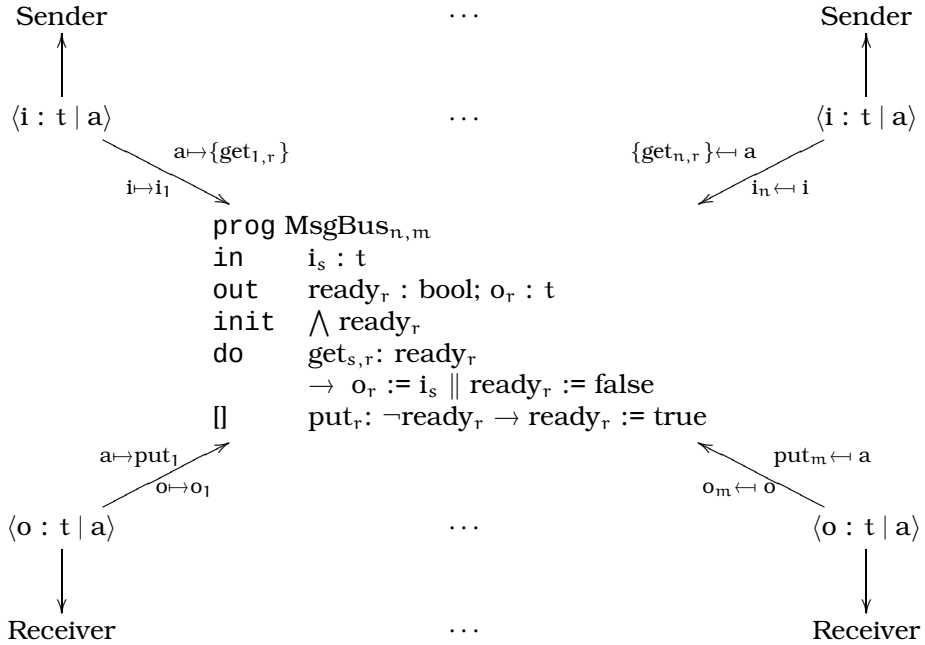

obtaining as a colimit

```
prog Unloading
in      idest, ibag : int
out     ready : bool; loc, odest, obag, gloc, bag : int; q : list(int)
init    ready ∧ 0 ≤ loc ≤ U-1 ∧ odest = -1 ∧ obag = 0 ∧ 0 ≤ gloc ≤ U-1 ∧ q = []
do      move: loc ≠ odest → loc := loc +U 1
[]      cart_get: odest = -1 → obag := ibag ‖ odest := idest
[]      put: loc = odest ∧ ready
        → obag := 0 ‖ odest := -1 ‖ bag := obag ‖ ready := false
[]      gate_get: ¬ready → q := [bag] + q ‖ ready := true
[]      move_gate_get: ...
[]      cart_get_gate_get: ...
```

□

If a receiver gets messages from different senders $s = 1, \ldots, n$, there are several possible assignments $M' := M_s$. Due to the locality principle, and to have well-formed diagrams, all assignments to a variable must be in a single program. Therefore for each kind of message a receiver might get, there is a single connector connecting it to all possible senders. On the other hand, a message might be sent to one of different receivers $r = 1, \ldots, m$. Therefore there will be several possible assignments $M'_r := M$ associated with the same 'put' action of the sender of message $M$. So there must be a single connector to link a sender with all its possible recipients. To sum up, for each kind of message there is a single connector acting like a data bus: at each step it transmits a message from one sender to one receiver.

**Definition 4.51.** The *message bus* connector for point-to-point transmission of a variable of sort $t$ between $n$ senders and $m$ receivers is




with $s = 1, \ldots, n$ and $r = 1, \ldots, m$. □

Allen and colleagues [AGI98] also use a single connector to provide communication between different parties. In particular they represent as a connector the whole run-time infrastructure of the High Level Architecture for Distributed Simulation, a component integration standard used by the US Department of Defense to support interoperability of simulations provided by different vendors.

### 4.6.3 Operations

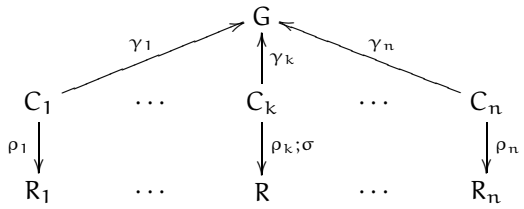
Garlan [Gar98] argues for principled ways of modifying connectors. His position is briefly summarized as follows. It is not always possible to adapt components to work with the existing connectors. Even in those cases where it is feasible, a better alternative might be to modify the connectors because usually there are fewer connector types than components types. Moreover, most Architecture Description Languages either provide a fixed set of connectors or only allow the creation of new ones from scratch, hence requiring from the designer a deep knowledge of the particular formalism and tools at hand. Conceptually, operations on connectors allow one to factor out common properties for reuse and to better understand the relationships between different connector types. The notation and semantics of such connector operators are of course among the main issues to be dealt with.

We feel that Category Theory is well suited to represent connectors and their construction at an abstract level independent of the formalism used to write specific connectors, thus revealing their fundamental properties. In this section we present four connector transformation operations. They are specified on roles since those form the interface of a connector, i.e., they are the only parts of the connector available to the user. This means that the user can form new connectors even from those to which he has no access to the implementation.

#### Role Refinement

The connectors presented in the previous section are general-purpose. To tailor them for a specific application, it is necessary to replace the generic roles by specialized ones that can effectively act as "formal parameters" for the application at hand. Role replacement is done in the same way as applying a connector to components: there must be a morphism from the generic role to the specialized one. The old role is canceled, and the new role morphism is the composition of the old one with the specialisation morphism.

**Definition 4.52.** Given an $n$-ary connector and a program morphism $\sigma : R_k \to R$ for $0 < k \leq n$, the *role refinement* operation yields the connector




□

*Example 4.19.* A unary connector

$$G \xleftarrow{\gamma} C \xrightarrow{\rho} R$$

can be refined with $R'$

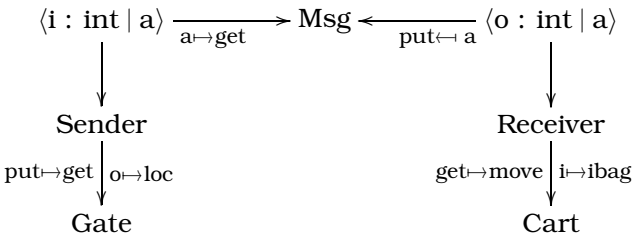
$$G \xleftarrow{\gamma} C \xrightarrow{\rho} R \xrightarrow{\sigma} R'$$

becoming the new connector

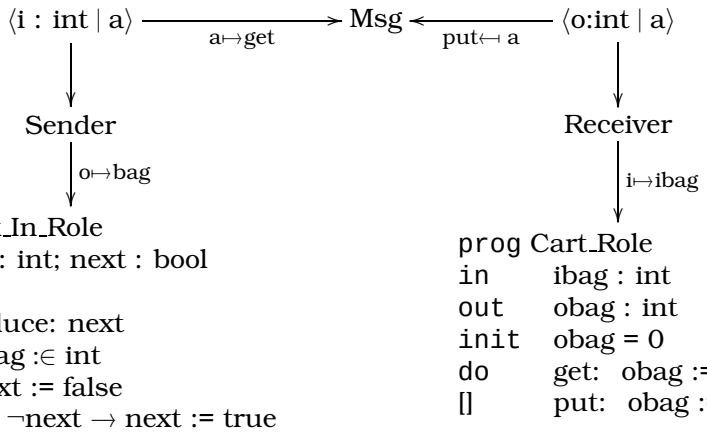
$$G \xleftarrow{\gamma} C \xrightarrow{\rho;\sigma} R'$$

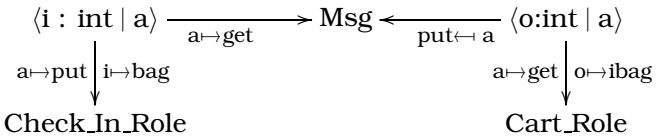
□

*Example 4.20.* The message passing connector can be applied in many ways to the programs for carts, check-in counters, and gates since roles 'Sender' and 'Receiver' do not impose any special constraint. Any signature morphism will do and result in a well-formed diagram. For example, here is a valid but completely meaningless application that sends the gate's location as a bag to the cart.




What we want is to allow only the three communications that take place in our example: a bag and a destination are transferred from a check-in counter to the cart and then the bag from the cart to the gate. We only show the connector needed to load a bag onto a cart and how to obtain it from the generic connector.



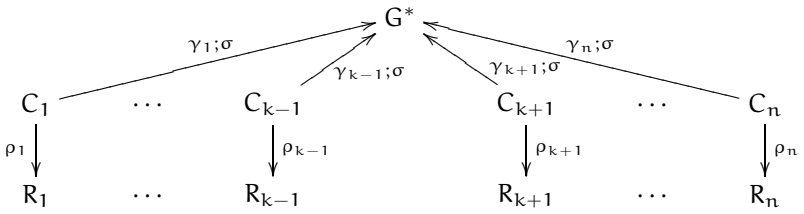




Notice that the given attributes and actions in the roles are indeed enough to discriminate among carts, check-in counters, and gates, thus guaranteeing that the roles are not applied to the wrong components. □

**Role Encapsulation**

To prevent a role from being further refined, the second operation we consider, when executed repeatedly, decreases the arity of a connector by encapsulating some of its roles, making the result part of the glue. In categorical terms, the glue of the new connector is the colimit of the diagram consisting of the old glue plus the encapsulated roles and the channels connected to them.

**Definition 4.53.** Given an $n$-ary connector the *role encapsulation* operation yields the connector of arity $n-1$

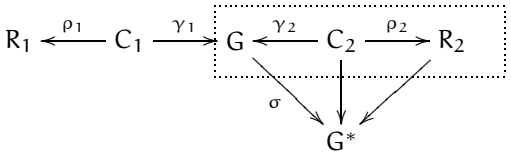



with $\sigma : G \to G^*$ and $G^*$ the colimit object of $G \xleftarrow{\gamma_k} C_k \xrightarrow{\rho_k} R_k$. □

*Example 4.21.* As an illustration, encapsulating the second role of a binary connector results in a unary connector.

1. original connector: $R_1 \xleftarrow{\rho_1} C_1 \xrightarrow{\gamma_1} G \xleftarrow{\gamma_2} C_2 \xrightarrow{\rho_2} R_2$

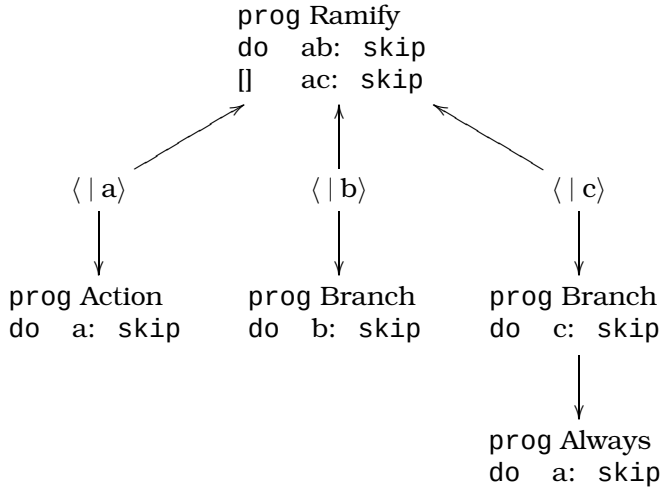
2. colimit calculation:



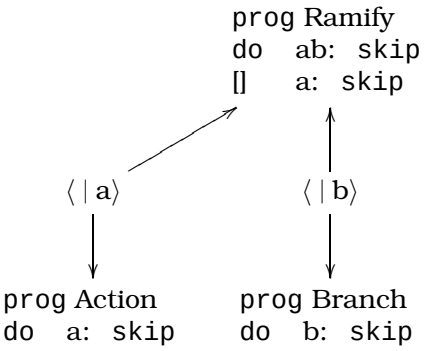

3. role encapsulation: $R_1 \xleftarrow{\rho_1} C_1 \xrightarrow{\gamma_1;\sigma} G^*$ □

This operation, combined with the previous one, shows how certain connectors are related.

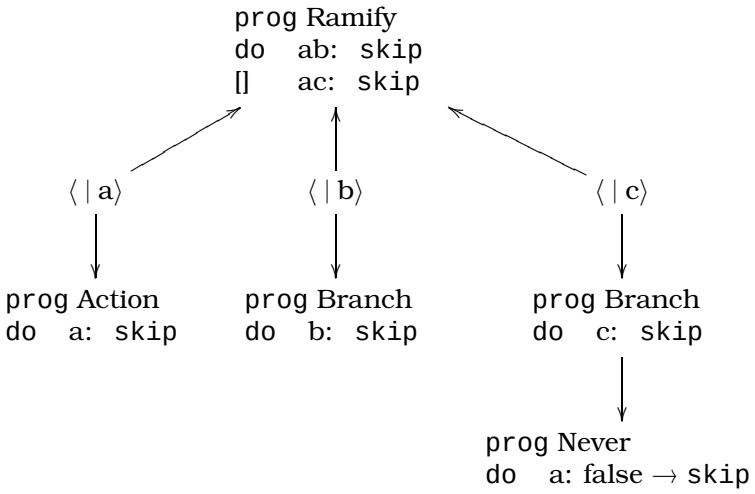
*Example 4.22.* We can obtain from the action ramification connector both the action subsumption and the synchronisation connectors, through refinement and then encapsulation of one of the 'Branch' roles. The refinement determines the obtained connector. From



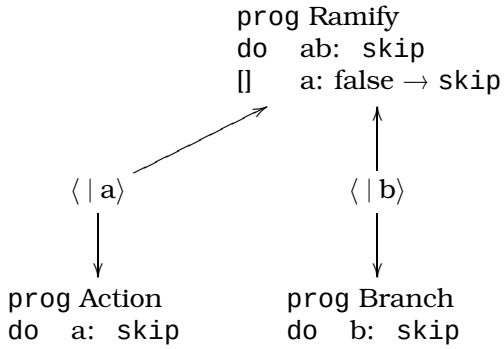
we get




and from



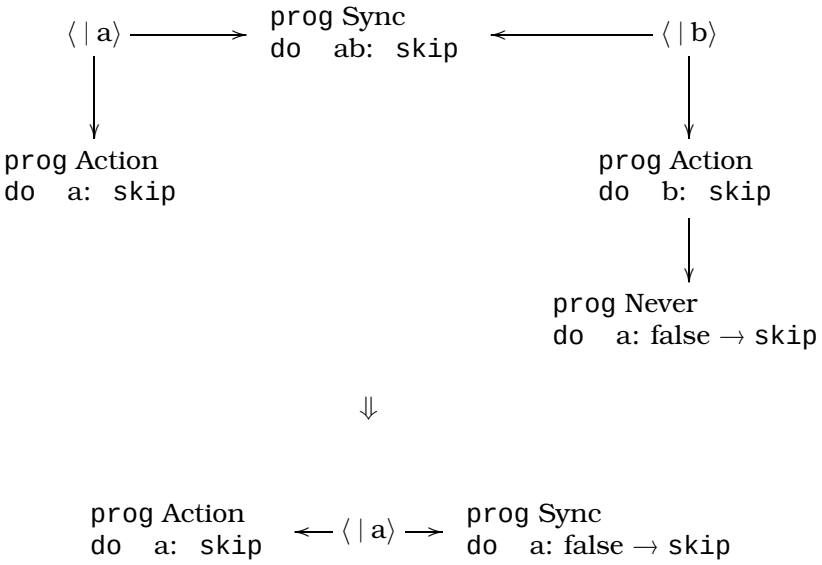

we obtain




Notice that the synchronisation connector obtained is not syntactically equal to the one presented in Definition 4.46 on page 75, but it is equivalent since the sub-action 'a' of the glue is never executed and thus does not influence when the action of the left role occurs. □

*Example 4.23.* If we apply the same technique to the synchronisation connector we ob-

tain the generic inhibition connector of Definition 4.49 on page 78.

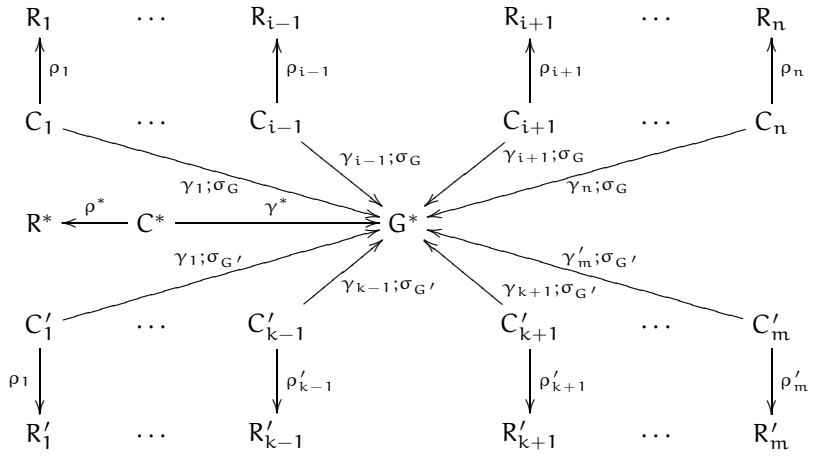



□

**Role Overlay**

The third operation allows to combine several connectors into a single one if they have some roles in common, i.e., if there is an isomorphism between those roles. Intuitively, the new connector is obtained as follows. First, take the diagrams of the original connectors and overlay them on the common roles. Then, remove the other roles and the corresponding channels. The glue of the new connector is the colimit of the remaining diagram. The new channel between the new glue and a common role is the colimit of the old channels.

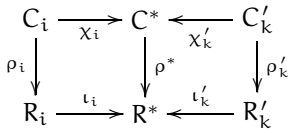
**Definition 4.54.** Given an $n$-ary connector with role $R_i$ and an $m$-ary connector with role $R'_k$ such that $\iota_i : R_i \to R^*$ and $\iota'_k : R'_k \to R^*$ are isomorphisms for some program $R^*$, the *role overlay* operation yields the $(n+m-1)$-ary connector




where $\{\sigma_X : X \to G^* \mid X \in \{G, G', C_i, C'_k, R_i, R'_k, R^*\}\}$ is the colimit of

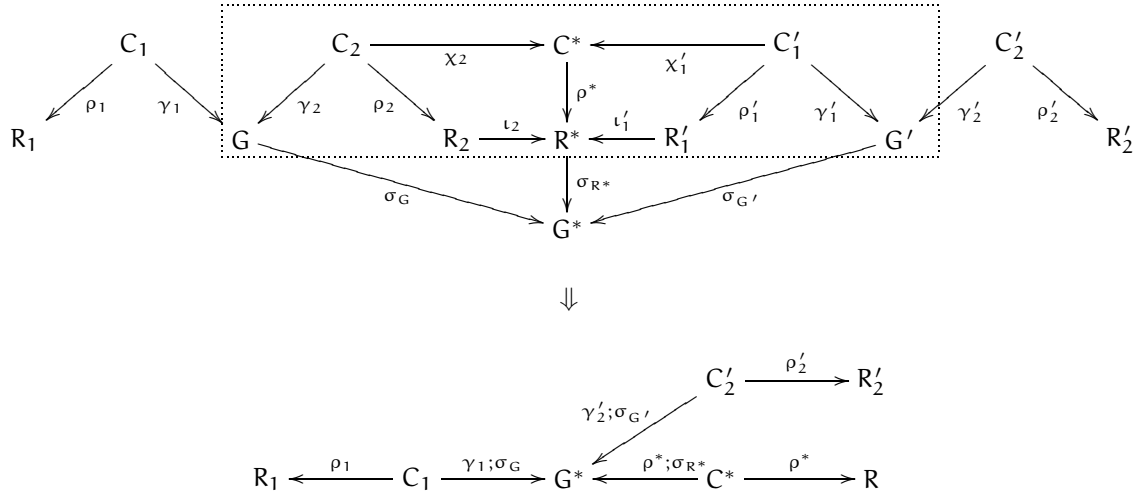
$$G \xleftarrow{\gamma_i} C_i \xrightarrow{\rho_i} R_i \xrightarrow{\iota_i} R^* \xleftarrow{\iota'_k} R'_k \xleftarrow{\rho'_k} C'_k \xrightarrow{\gamma'_k} G',$$

$\chi_i : C_i \to C^*$ and $\chi'_k : C'_k \to C^*$ is the coproduct of $C_i$ and $C'_k$, $\gamma^* = \rho^*; \sigma_{R^*}$, and $\rho^*$ makes




commute. □

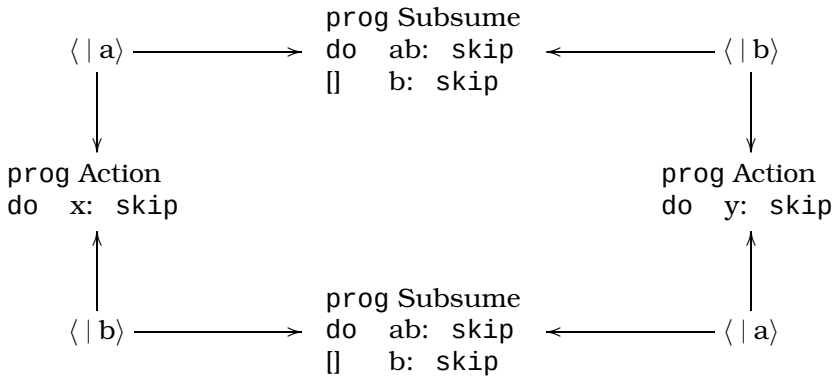
*Example 4.24.* As an illustration, consider the overlay of two binary connectors (i.e., $m = n = 2$) on one common role. In particular, $i = 2$ and $k = 1$.



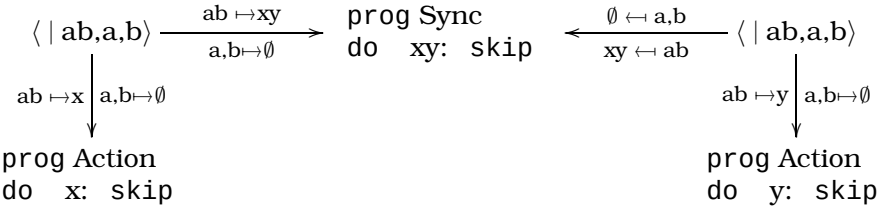

□

As the example makes clear, $R^*$ and $G^*$ are cocone objects of $C_i$ and $C'_k$ and therefore $\rho^*$ and $\gamma^*$ exist and are unique.

*Example 4.25.* This operation provides a second way of showing that synchronisation is not a primitive connector within our catalog. We can indeed obtain full synchronisation of actions 'a' and 'b' by making 'a' subsume 'b' and vice-versa. This is achieved by overlaying two copies of the basic connector in a symmetric way: the first (resp. second) role of one copy corresponds to the second (resp. first) role of the other copy. The diagram is

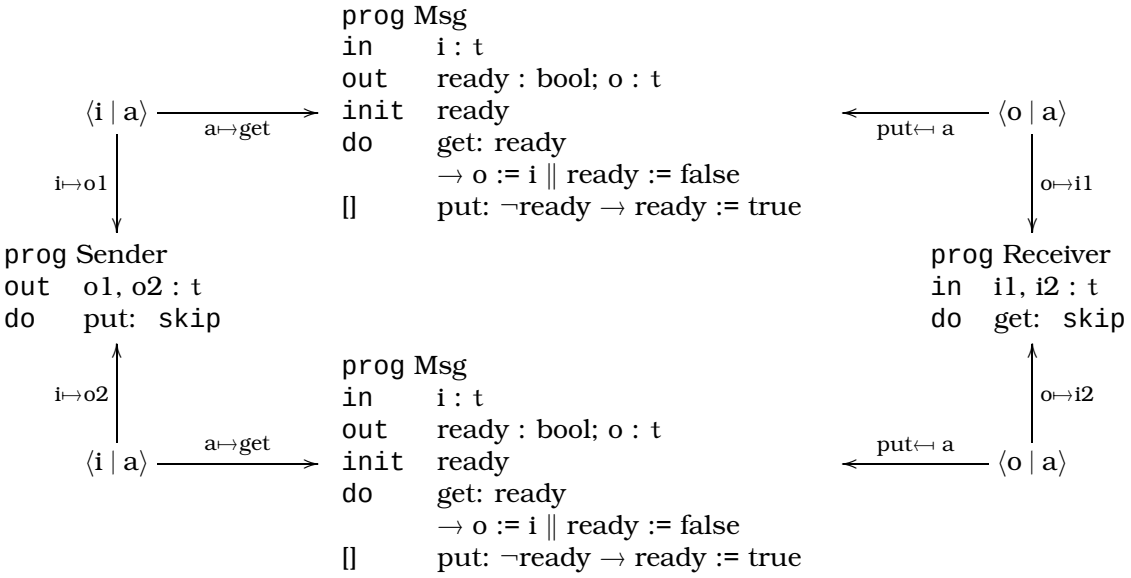



and the resulting connector is

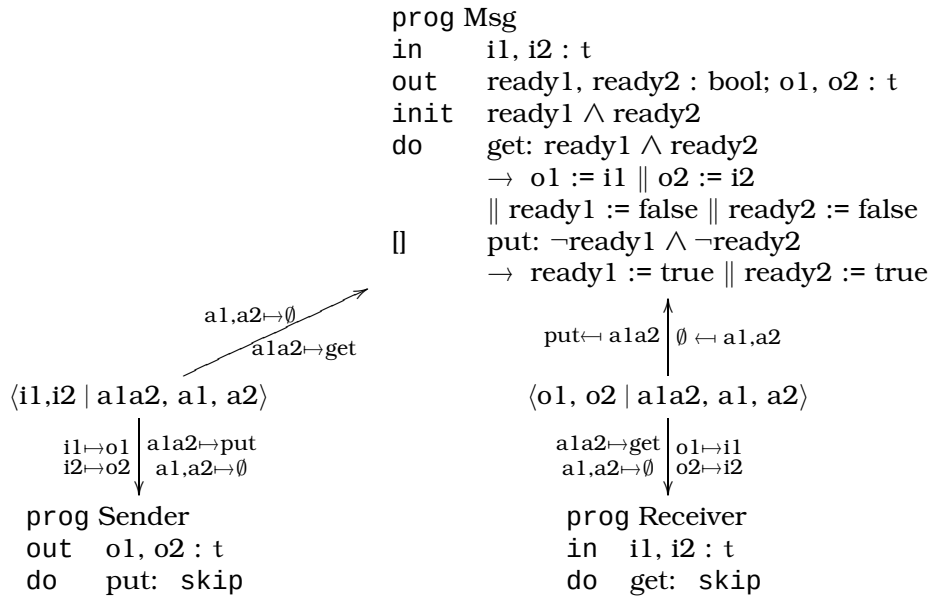



Again, the result is not equal to the original one, but the extra actions in the channels are superfluous as they do not establish any extra links between roles and glue. □

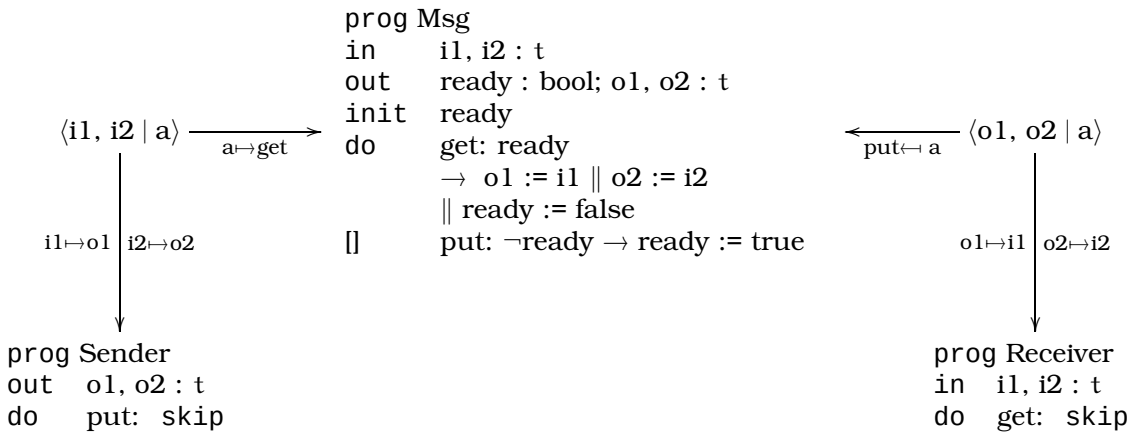
*Example 4.26.* To obtain a connector that transmits two variables of the same sort t, we take two copies of the message passing connector, refine their roles with the same programs but using different morphisms, and then overlay them obtaining the diagram




with the resulting connector



Since the two 'ready' variables are initialised and modified in the same way, and because the extra channel actions are superfluous, an equivalent connector is




□

**Transient Connectors**

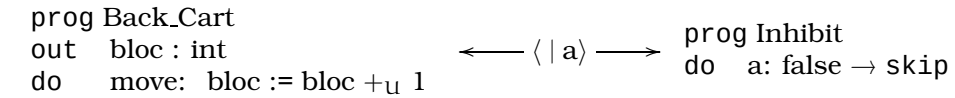
So far none of the connectors presented handles transient interactions, the bread and butter of our example. To do so, we introduce transient connectors, which is a shorthand notation to specify connectors that are active only when a certain condition holds. That condition may depend not only on the state of the interacting components but also on the state of others which, although not taking part in the interaction, provide the context for it to happen.

**Definition 4.55.** An $m$-*ary transient connector* is a triple $\langle C, \{R_i\}_{n<i\leq m}, ac\rangle$ where $C$ is a $n$-ary connector with $n \leq m$, $R_i$ are programs called roles, and $ac \in Props(\bigcup_{j=1,\ldots,m} V_j)$ is called the *application condition*, where $V_j$ are the variables of the $j$-th role. The sets $V_{1,\ldots,m}$ must be mutually disjoint. □

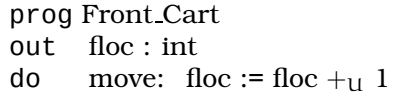
Notice that the definition imposes the variables of the roles to be distinct. Otherwise it is not possible to know to which ones the application condition refers.

To illustrate the need for additional programs that only provide the interaction context, we present an alternate way of preventing collisions: a cart must stop while there is a cart in the next track unit.

*Example 4.27.* A transient inhibition connector to prevent two carts from colliding is $\langle \text{Inhibit}, \text{Front\_Cart}, \text{floc} = \text{bloc} +_u 1\rangle$ with




and




□

We can now present our last operation, which provides the semantics of a transient connector by transforming it into a "regular" connector. The key ideas are:

1. the actions of the glue are only valid when the application condition is true,
2. each action of a role is divided into two sub-actions, one occurring when the connector is inactive (i.e., $ac$ is false), the other in the opposite case.

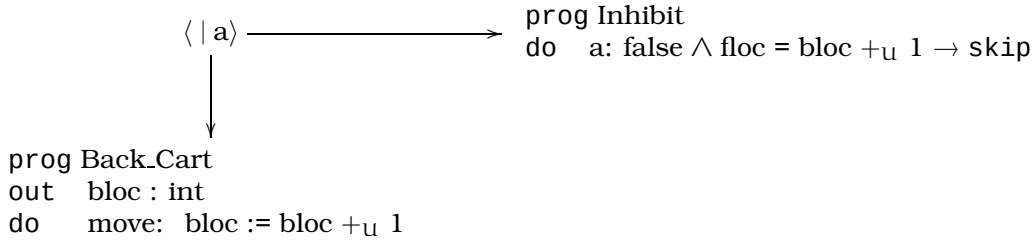
The latter sub-action is already included in the glue, so we must only add a new action for the former case. The glue of the resulting connector must of course declare all variables that occur in $ac$ (since the condition is now internalised in the glue's action guards), and the channels and their morphisms must show from which role each variable and each new action of the glue comes.

**Definition 4.56.** The *denotation* of a transient connector $\langle C, \{R_i\}_{n<i\leq m}, ac\rangle$ is given by a connector obtained as follows:

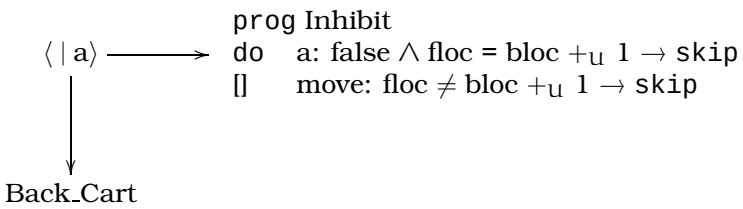
1. strengthen the glue's guards with $ac$;
2. for each role action $a$ that takes part in the interaction (i.e., is synchronized with a glue action), add $a$: $\neg ac \to$ `skip` to the glue;
3. add the roles $R_i$ and the empty channels $C_i$ (with $i = n+1, \ldots, m$);
4. add as necessary $ac$'s variables to the glue's input variables,
5. and to the necessary channels;
6. augment the morphisms accordingly. □

*Example 4.28.* We apply the prescribed steps to the transient connector in the previous example:

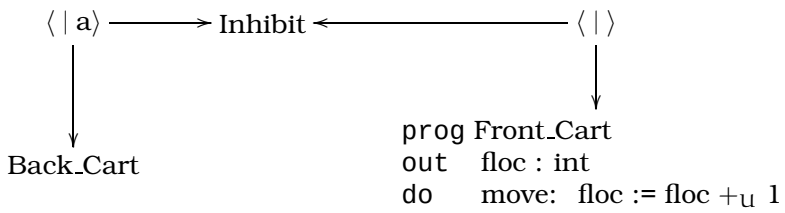
1. strengthen the glue's actions with $ac$;



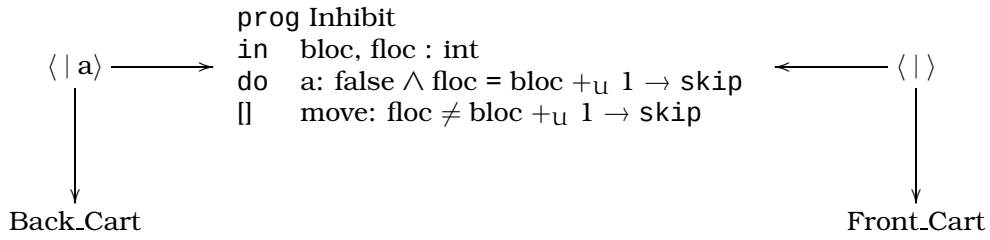

2. for each role action $a$ that takes part in the interaction (i.e., is synchronized with a glue action), add $a$: $\neg ac \to$ `skip` to the glue;



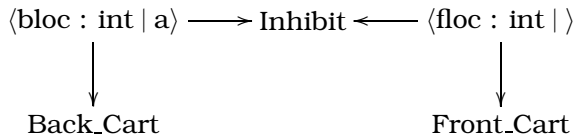

3. add the roles $R_i$ and the empty channels $C_i$ (with $i = n+1, \ldots, m$);



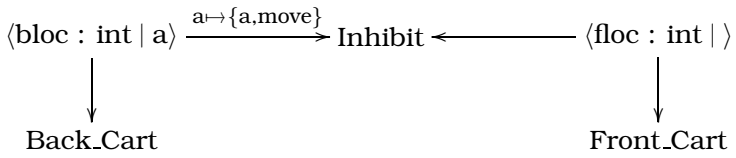
4. add as necessary ac's variables to the glue's input variables,




5. and to the necessary channels;




6. augment the morphisms accordingly.

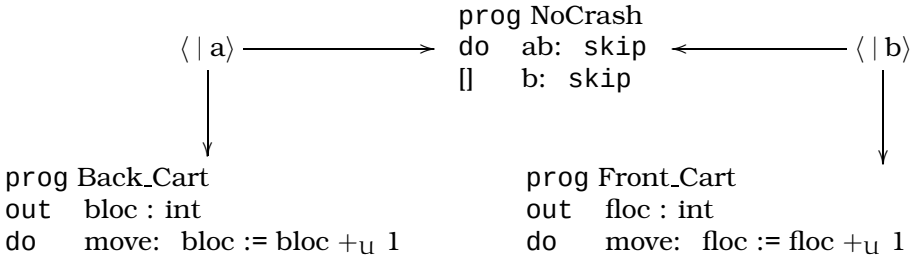



Due to our notation conventions, the last step seems to do almost nothing in this case. □

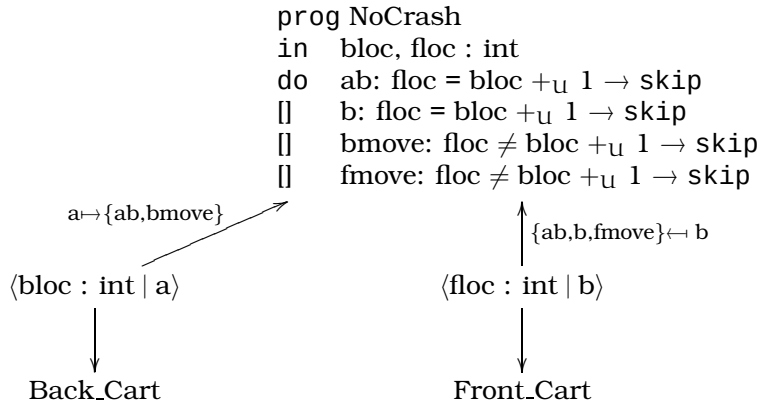
It is important to compare the original connector with the generated one to appreciate what is being gained. The most obvious difference is that the application condition is specified outside the connector, more precisely, outside the glue of the connector. This triggers some simplifications. With regular connectors, the glue, the channels, and their morphisms have to contain all the relevant variables from the roles because the application condition is inside the glue. This is not necessary with transient connectors because the condition is stated directly in terms of the variables of the roles. This means that, apart from making sure that attributes in different roles have different names, writing connectors becomes simpler (and thus less error-prone).

The second disadvantage is that the condition, being coded into the guards of the glue's actions in order to show explicitly when they can be executed, can only be changed if one has access to the implementation of the glue. In other words, without transient connectors it is not possible to provide libraries of pre-defined connectors to be (re)used for many applications or in different situations.

*Example 4.29.* The connector that we use to avoid collisions is obtained from the action subsumption connector in Definition 4.47 on page 76, refining its roles for carts, and then adding the condition on the proximity of the carts' locations, similarly to Example 4.27 on page 88. The transient connector is $\langle \text{NoCrash}, \emptyset, \text{floc} = \text{bloc} +_u 1 \rangle$ with

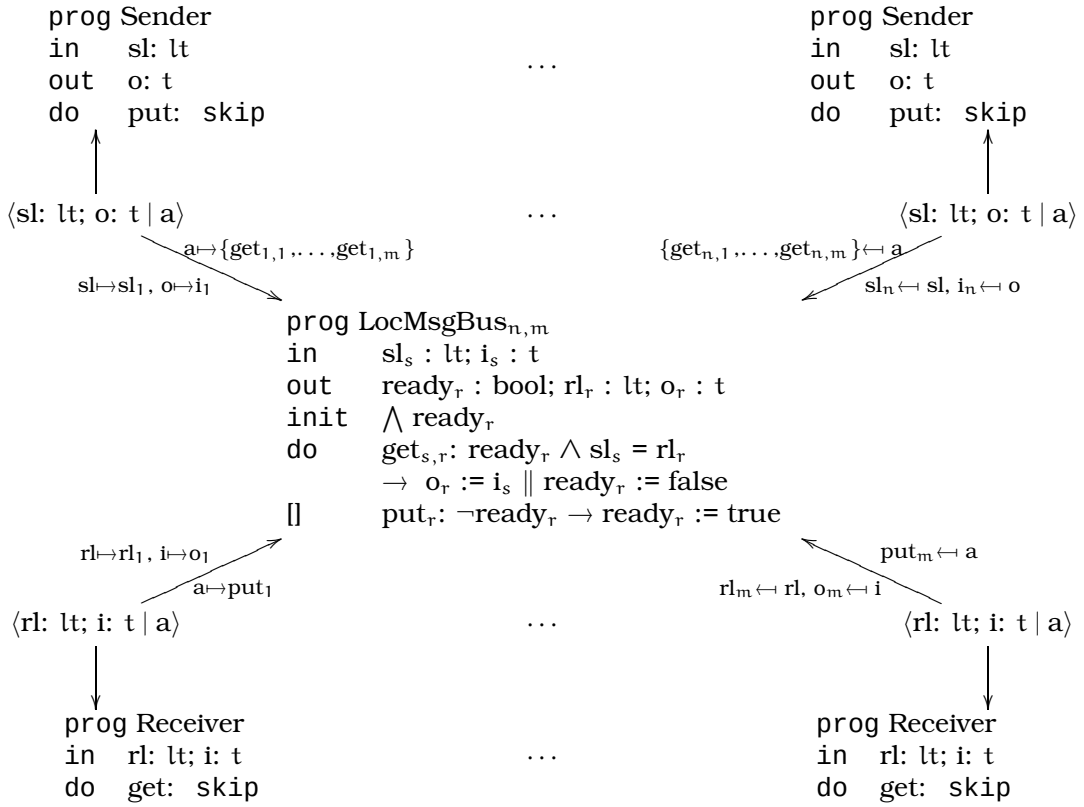


The resulting connector is




□

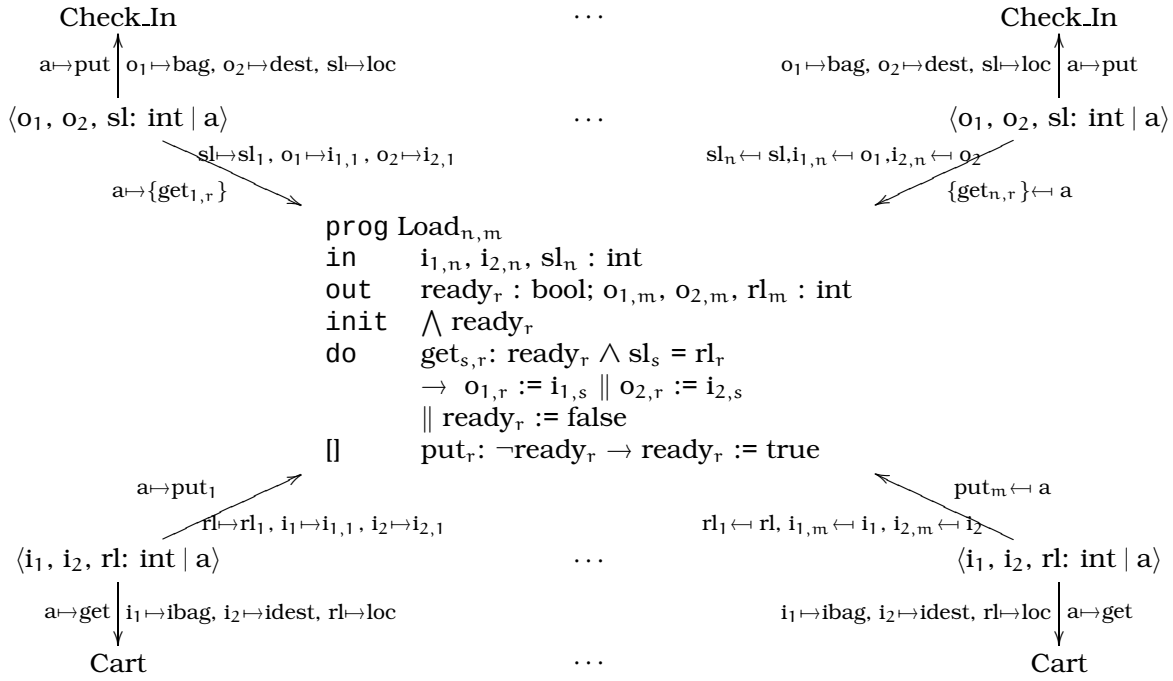
*Example 4.30.* Consider that senders and receivers communicate only when co-located. Therefore different pairs of sender/receiver interact under different conditions, and there is no automatic transformation mechanism as for the previous connectors in this section. Each guard of the glue in Definition 4.51 on page 80 must be individually changed. We must also add a variable to each sender and receiver to represent its location (of some arbitrary type lt).




with $s = 1, \ldots, n$ and $r = 1, \ldots, m$. □

*Example 4.31.* The connector 'Unload$_{n,m}$' to let $n$ carts unload bags at $m$ gates is the connector in the previous example with each sender role refined by a cart program and each receiver role refined by a gate and with lt = t = int.

The connector that allows $m$ carts to get bags and their destinations from $n$ check-in counters is obtained from 'LocMsgBus$_{n,m}$' using the same technique as in Example 4.26 on page 87. Furthermore, each sender role is refined with a check-in program and each receiver role with a cart. Finally, $t = lt = int$.




with $s = 1, \ldots, n$ and $r = 1, \ldots, m$. □

## 4.7 Architectures

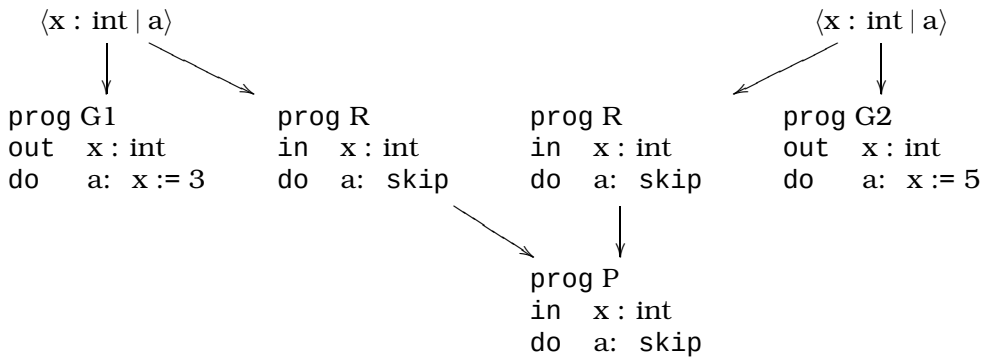
An architecture is a configuration where components interact through connectors, not individual channels. Moreover, all connectors appearing in the architecture are applied; there must not be any "dangling" roles. The definition is extended to program and connector instances, as usual. Being a configuration (instance), it follows that any architecture (instance) has a semantics given by its colimit.

**Definition 4.57.** An *architecture* is a configuration made of a multiset of programs and a multiset of connectors, such that each role is refined by some program. An *architecture instance* is a diagram D in $\mathfrak{Inst}$ such that $\mathcal{IP}(D)$ is an architecture. The *system (instance)* described by an architecture (instance) is given by its colimit. □

It should be noted that to check whether an architecture is indeed well-formed it is not enough to check that it is formed by applied connectors, which must be well-formed by Definition 4.43 on page 73, because the condition on paths between output variables in the data view is a global restriction.

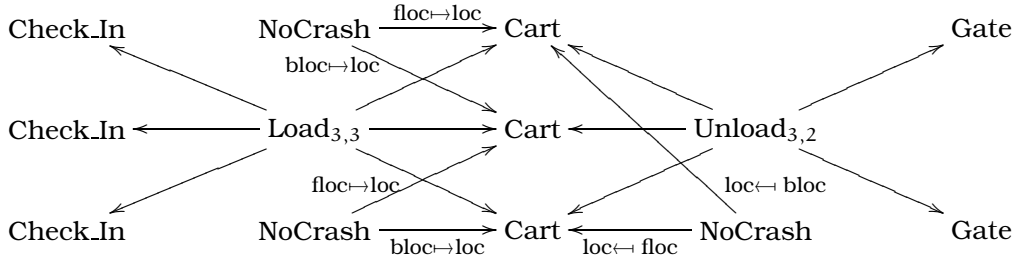
*Example 4.32.* The following diagram is not an architecture, although each unary con-

nector is correctly applied.




□

*Example 4.33.* The architecture of a luggage distribution system with three carts (Example 4.3 on page 61), three check-in counters (Example 4.5 on page 62), and two gates (Example 4.6 on page 62) is given by the following black-box view of the application of the connectors of Example 4.29 on page 90 and Example 4.30 on page 91.




The omitted mappings are identities. □

The only interesting architecture instance for system specification is the one that provides the initial values for each variable. For that purpose, each output variable is associated to a ground term such that the initialisation condition is satisfied.

**Definition 4.58.** An *initial architecture instance* is an architecture instance such that for each program instance $\langle P, \epsilon \rangle$, $\epsilon : O \to \mathcal{T}erms(\emptyset)$ and $\models_\emptyset \epsilon(ic)$. □

### 4.7.1 Style

In the previous chapter—more precisely in Section 3.2 on page 29—we viewed an architectural style as a language, the architectures conforming to the style being the words of that language. As such, a style was specified by a grammar that generated architectures. This view can be straightforwardly adapted to architectures of CommUnity programs using the algebraic approach to graph grammars presented in Section A.3 on page 116: a style is a graph grammar that generates diagrams in $\mathfrak{Prog}$ from a given start diagram (representing the start symbol of the grammar). As mentioned in the last chapter, taking a slightly different perspective, we may view a style as the set of all possible reconfigurations of the start diagram, making the specification of styles and reconfigurations similar. We present the algebraic approach to reconfiguration in the next section. Hence, there is no point in pursuing again this "style as language" approach here.

Instead, we take a declarative view of a style as a type. To be more precise, since an architecture is given by a graph, a style is a graph of types, and checking that

an architecture conforms to a style amounts to find a graph morphism mapping the former to the latter. A similar technique has been used in practice to check whether an implementation drifted from the design architecture [GAK99].

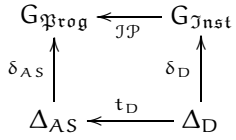
In the previous approach it is necessary to prove explicitly that the reconfiguration rules do not generate architectures that do not belong to the style, while in the typing approach this is automatically enforced. On the other hand, the graph grammar approach to style is more expressive than the typed graph approach. For example, it allows to impose constraints on the number of components. However, we believe that typed graphs are sufficient, simpler, and more straightforward in many occasions, namely when only the kinds of interactions between the components have to be restrained.

This is achieved by restricting the ways the connectors can be applied to components. In general, a role may be instantiated by different components, and it may be even the case that the same component can instantiate the same role in different ways. But normally those cases are rather pathological and we want to rule them out of the meaningful architectures. Furthermore, there may be "functional dependencies" between the roles. For example, each role of a binary connector may be instantiated with programs $P_{1,2,3}$ but we want to impose the constraint "if the first role is applied to program $P_i$, then the second role is refined with $P_{4-i}$". Sometimes such restrictions can be obtained by fine-tuning the roles until they can be refined only by the intended programs with the intended morphisms. However, even if possible, it may not be desirable to do so because the obtained connectors are less reusable, less understandable, and the process is error-prone, requiring validation.

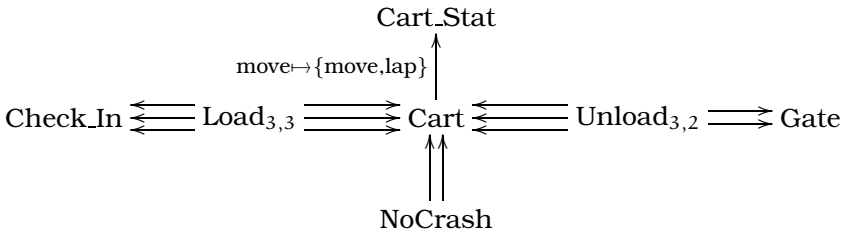
It is easier to describe the constraints using typed graphs. This leads to a declarative notion of architecture style: it is a diagram $AS$ stating how the connectors and components are to be used. For the mentioned example, $AS$ would contain three copies of the binary connector applied to the the pairs of programs $P_1/P_3$, $P_2/P_2$ and $P_3/P_1$. Every architecture written by the user must then come equipped with a morphism to $AS$ proving that it obeys the restrictions imposed by the style.

**Definition 4.59.** An *architectural style* $AS$ is a diagram in $\mathfrak{Prog}$ using only the existing connectors and components. An architecture (instance) $D$ *conforming* to a style $AS$, also called *$AS$-architecture (instance)*, is a pair $\langle D, t_D \rangle$ with $t_D$ a labelled graph morphism from $D$ (resp. $\mathcal{IP}(D)$) to $AS$. □

The morphism $t_D$ guarantees that the typing is meaningful, e.g., a cart in the architecture will not be typed by a gate in the style. If $\langle D, t_D \rangle$ is an architecture instance conforming to a style $AS$, then the following diagram in $\mathfrak{Graph}$ commutes.




*Example 4.34.* Consider the following style for the architecture shown previously, using again the black-box notation.

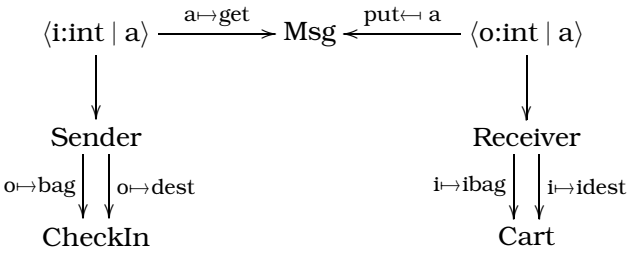


This diagram basically states that carts may be refined by carts with counters, all sender roles of the 'Unload$_{3,3}$' connector must be applied to counters, all receiver roles are applied to carts, etc. This example also illustrates that an actual architecture does not have to use all connectors and components given by the style. □

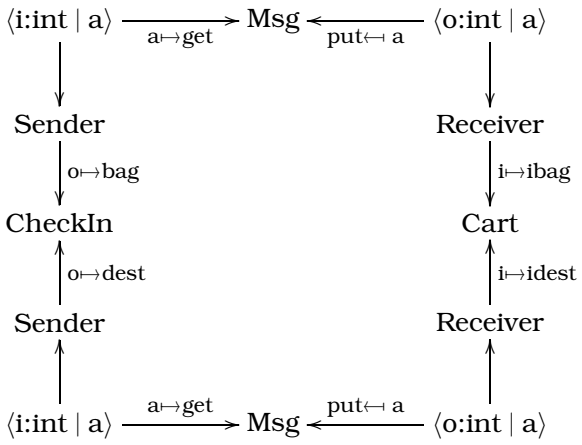
This style provides two kinds of information. First, it constrains what components each connector can be applied to. For example, it states that the action subsumption connector is only to be used for carts' movement; it prevents the 'put' action of a counter to subsume the 'get' action of a gate, among other combinations. Second, it constrains the visibility of a program's variables and actions. For example, it states that the 'new' action and the 'q' variable of a check-in station (Example 4.5 on page 62) are only to be used by it. In [Lop99] private output variables and private actions were introduced. As the name implies, they may not be shared or synchronised with variables or actions of other programs. This leads to several changes in the definitions of COMMUNITY. With our approach, the style simply does not include any morphism involving a variable or action that is to be private. However, from a practical and conceptual point of view, it is better to let the programmer state as early as possible which variables and actions are intended only for local computations done by the program, instead of delaying it to the style specification. Moreover, the execution of private actions can obey fairness constraints, which cannot be captured by our notion of style.

In our categorical framework, an architectural style represents all combinations of possible morphisms between roles and components. Hence, styles capture a finer-grained notion than role dependency, they represent morphism dependency. The style specifier must take care not to put between a given connector and set of components morphisms that must not occur together.

*Example 4.35.* The following two style excerpts are not equivalent, i.e., the sets of conforming architectures are distinct: whereas




allows a bag to be passed as a destination (and vice-versa),

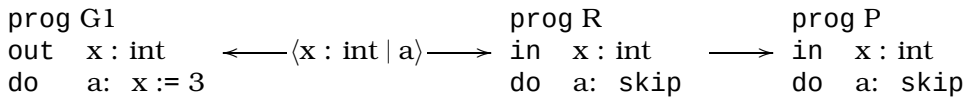



does not. □

In the definition we have not required the style to be an architecture, i.e., the style does not have to be well-formed nor obey the restrictions on degrees stated in Definition 4.33 on page 68. The reason is that a style shows all possible morphisms in one single diagram. For example, there may be several refinement morphisms from the same role to the several components to which it may be applied, thus violating the constraint on having outdegree one. If the morphisms map an input variable of the role to output variables of the components, the diagram is not well-formed either.

To compensate for the lack of flexibility, there would have to be an added value in requiring a style to be well-formed, namely to force any conforming architecture to be well-formed. However, that cannot be achieved.

*Example 4.36.* Well-formedness is not preserved by typed graph morphisms. Imagine the diagram of Example 4.32 on page 92 with both glues being 'G1'. The two output variables would still be indirectly shared, but the diagram could be typed by the well-formed diagram




□

## 4.8 Reconfiguration

Our treatment of reconfiguration so far has two drawbacks. First, the specification may grow quite large because the architecture of the system is given by a fixed diagram showing *all* possible connections between the existing components, even if due to the actual computations some components will never interact. In our example, the diagram for an architecture with c carts has c copies of the action subsumption connector. The other disadvantage is that connectors have to capture all transient interactions and thus may become quite complex and specific to the number of interacting components. Our message bus illustrates this point.

The semantics we present in this section instead captures the dynamic flavour of the configurations in a more explicit way by formalising transient connectors as conditional rewrite rules over architecture instances.

### 4.8.1 Rules

Since an architecture is a diagram in a given category, which in turn is a graph typed over the objects and morphisms of that category, the algebraic graph transformation approach summarised in Section A.3 on page 116 can be directly applied to architecture reconfiguration.

**Definition 4.60.** A *reconfiguration rule* is a graph production typed over $G_{\mathfrak{Prog}}$ where L, K, and R are architectures. □

Whereas the relational graph grammars of Section 3.2 on page 29 encode graphs, the algebraic approach manipulates directly the mathematical structures. It thus guarantees that the architecture is at all times during reconfiguration a graph (in particular a diagram), a necessary condition for our approach, based on colimits. The relational characterization of graphs and their transformation cannot enforce that (see Section 3.3.1 on page 31).

When a production only adds nodes and arcs, it may be reapplied again immediately because the left-hand side is a sub-graph of the right-hand side. If the left-hand side

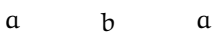
is matched more than once to the same part of the graph to be rewritten, then no real new information is being added. Moreover, this leads to infinite rewriting sequences. We thus restrict the allowed derivations.

**Definition 4.61.** A direct derivation $G \overset{p,m}{\Longrightarrow} H$ (typed over a graph TG) is called *productive* if there are no (typed) morphisms $lr : L \to R$ and $x : R \to G$ such that $lr; x = m$. □

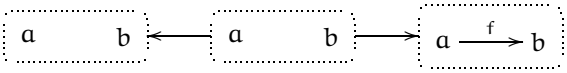
The existence of morphism $lr$ shows that the production does not delete any nodes or arcs. The remaining conditions check that the match is being applied to a part of $G$ that corresponds to the right-hand side and therefore can have been generated by a previous application of this production.

*Example 4.37.* A production of the form $L \longleftarrow \emptyset \longrightarrow L$ can never be used in a productive derivation. □

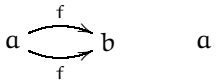
*Example 4.38.* Consider the labelled graph without edges



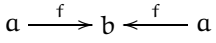

and the production




A sequence of two direct derivations might lead to the graph




whereas a sequence of two productive derivations can only result in




and then no further productive derivation is possible. □

Our definition is a particular case of productions with application conditions in the sense of [HHT96]: a derivation $G \overset{p,m}{\Longrightarrow} H$ is productive if $p$ is applicable to $G$ using the negative application condition $lr$.

We can now define a reconfiguration step as a productive direct derivation from a given architecture $G$ to an architecture $H$. In the algebraic graph transformation approach, there is no restriction on the obtained graphs, but in reconfiguration we must check that the result is indeed an architecture, otherwise the rule (with the given match) is not applicable. For example, two separate connector addition rules may each be correct but applying them together may yield a diagram which is not an architecture (see Example 4.32 on page 92).

**Definition 4.62.** A *reconfiguration step* is a productive direct derivation $G \overset{p,m}{\Longrightarrow} H$ typed over $\mathfrak{Prog}$ where $p$ is a reconfiguration rule and $G$ and $H$ are architectures. □

If there is an architectural style, then the three architectures in a reconfiguration rule must conform to the style and the morphisms between them must also preserve the typing given by the style.

**Definition 4.63.** An *AS-reconfiguration rule* for a style $AS$ is of the form $p_{AS} : (\langle L, t_L\rangle \xleftarrow{l} \langle K, t_K\rangle \xrightarrow{r} \langle R, t_R\rangle)$ where

- $\{\langle X, t_X\rangle \mid X = L, K, R\}$ are $AS$-architectures,

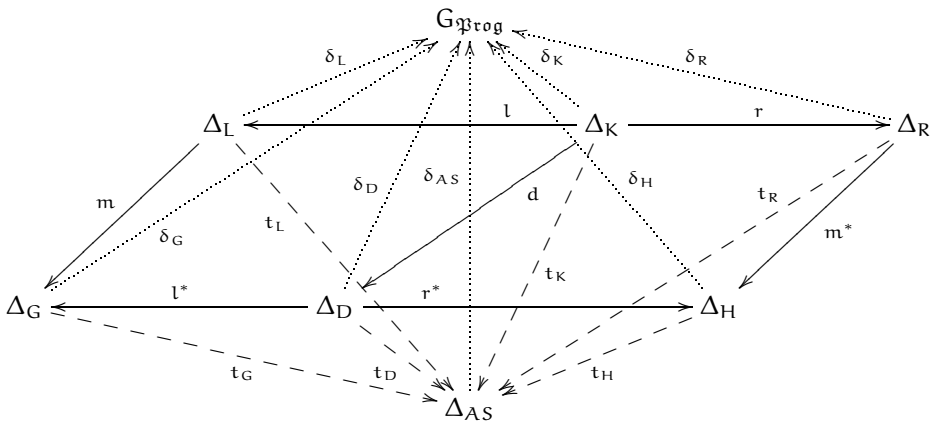
- $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ is a reconfiguration rule,
- $l; t_L = t_K = r; t_R$. □

A rule conforming to a given style can only be applied to architectures conforming to the same style, and the match must preserve the typing given by the style. This guarantees that the resulting architecture (if it exists) also conforms to the style.

**Definition 4.64.** Given a style $AS$, an $AS$-architecture $\langle G, t_G \rangle$ and an $AS$-reconfiguration rule $p_{AS}$, an *$AS$-reconfiguration step* $\langle G, t_G \rangle \overset{p_{AS},m}{\Longrightarrow} \langle H, t_H \rangle$ is a reconfiguration step $G \overset{p,m}{\Longrightarrow} H$ with $m; t_G = t_L$. □

**Proposition 4.15.** *The result of an reconfiguration step conforming to a style $AS$ is always an $AS$-architecture $\langle H, t_H \rangle$ with unique $t_H$.* □

*Proof.* Consider the following diagram in $\mathfrak{Graph}$




The morphism $t_D$ exists and is unique because $\langle \Delta_D, t_D \rangle$ is the pushout complement object in $(\mathfrak{Graph} \downarrow \Delta_{AS})$. Similarly for $t_H$. We now prove $\langle D, t_D \rangle$ is an $AS$-architecture:

$$\begin{aligned} t_D; \delta_{AS} &= l^*; t_G; \delta_{AS} && \text{Proposition A.5 on page 115 for } (\mathfrak{Graph} \downarrow \Delta_{AS}) \\ &= l^*; \delta_G && \langle G, t_G \rangle \text{ is an } AS\text{-architecture} \\ &= \delta_D && \text{Proposition A.5 on page 115 for } (\mathfrak{Graph} \downarrow G_{\mathfrak{Prog}}) \end{aligned}$$

As for $\langle H, t_H \rangle$:

$$\begin{aligned} d; t_H; \delta_{AS} &= t_D; \delta_{AS} && \text{Proposition A.4 on page 114 for } (\mathfrak{Graph} \downarrow \Delta_{AS}) \\ &= \delta_D && \langle D, t_D \rangle \text{ is an } AS\text{-architecture} \\ &= d; \delta_H && \text{Proposition A.4 on page 114 for } (\mathfrak{Graph} \downarrow G_{\mathfrak{Prog}}) \end{aligned}$$

From the uniqueness it results $\delta_H = t_H; \delta_{AS}$. ✓

Dynamic reconfiguration basically is a rewriting process over architecture instances, i.e., graphs labelled with program instances instead of just programs. This ensures that reconfiguration and computation are kept separate because the state of components and connectors that are not deleted nor added by a rule does not change, due to the preservation of labels enforced by typed graph morphisms. As for components introduced by the rule, we provide full control to the rule writer, letting him specify exactly in which state new components are added to the architecture. For that purpose we require that the logical variables occurring on the right-hand side of a rule also occur on the left-hand side.

Furthermore, dynamic reconfiguration rules depend on the current state. Thus they must be conditional rewrite rules. Within the algebraic graph transformation framework it is possible to define conditional graph productions in a uniform way, using only graphs and graph morphisms [HHT96]. However, for our representation of components it is simpler, both from the practical and formal point of view, to represent conditions as boolean expressions over logical variables.

**Definition 4.65.** A *dynamic reconfiguration rule* $\langle p, mc \rangle$ is a graph production $p$ typed over $G_{\mathfrak{Inst}}$ where $L$, $K$, and $R$ are architecture instances, $Vars(R) \subseteq Vars(L)$, and $mc \in Props(Vars(L))$ is the *matching condition*. □

The definition of reconfiguration step must be changed accordingly. At any point in time the current system is given by an architecture instance without logical variables. Therefore the notion of matching must also involve a compatible substitution of the logical variables occurring in the rule by ground terms. Applying the substitution to the whole rule, we obtain a rule without logical variables whose left hand side can be directly matched to the current architecture. The reconfiguration proceeds as a normal derivation (i.e., as a double pushout over typed graphs). However, the notion of state introduces two constraints. First, the substitution must obviously satisfy the matching condition. Second, the state of each program instance added by the right-hand side satisfies the respective initialisation condition.

**Definition 4.66.** Given a dynamic reconfiguration rule $\langle p, mc \rangle$, an architecture instance $G$, and a substitution $\phi : Vars(L) \to Terms(\emptyset)$, a *dynamic reconfiguration step* is a productive direct derivation $G \overset{\phi(p),m}{\Longrightarrow} H$ typed over $G_{\mathfrak{Inst}}$ such that

- $\phi(p)$ is the rule obtained through replacement of every program instance $\langle P, \epsilon \rangle$ by $\langle P, \epsilon' \rangle$, with $\epsilon'(o) = \phi(\epsilon(o))$ for every $o \in O$,
- $\models_\emptyset \phi(mc)$,
- for each $\langle P, \epsilon \rangle$ in $R \setminus r(K)$, $\models_\emptyset \phi(\epsilon(ic))$. □

The definitions are trivially adapted to handle styles.

**Definition 4.67.** An *AS-dynamic reconfiguration rule* for a style $AS$ is a pair $\langle p_{AS} : (\langle L, t_L \rangle \xleftarrow{l} \langle K, t_K \rangle \xrightarrow{r} \langle R, t_R \rangle), mc \rangle$ where

- $\{\langle X, t_X \rangle \mid X = L, K, R\}$ are $AS$-architecture instances,
- $\langle p : (L \xleftarrow{l} K \xrightarrow{r} R), mc \rangle$ is a dynamic reconfiguration rule,
- $l; t_L = t_K = r; t_R$. □

**Definition 4.68.** Given a style $AS$, an $AS$-architecture instance $\langle G, t_G \rangle$, an $AS$-dynamic reconfiguration rule $p_{AS}$, and a substitution $\phi : Vars(L) \to Terms(\emptyset)$, an *AS-dynamic reconfiguration step* $\langle G, t_G \rangle \overset{\phi(p_{AS}),m}{\Longrightarrow} \langle H, t_H \rangle$ is a reconfiguration step $G \overset{\phi(p),m}{\Longrightarrow} H$ with $m; t_G = t_L$. □

**Proposition 4.16.** *The result of a dynamic reconfiguration step conforming to a style* $AS$ *is always an* $AS$*-architecture instance* $\langle H, t_H \rangle$ *with unique* $t_H$. □

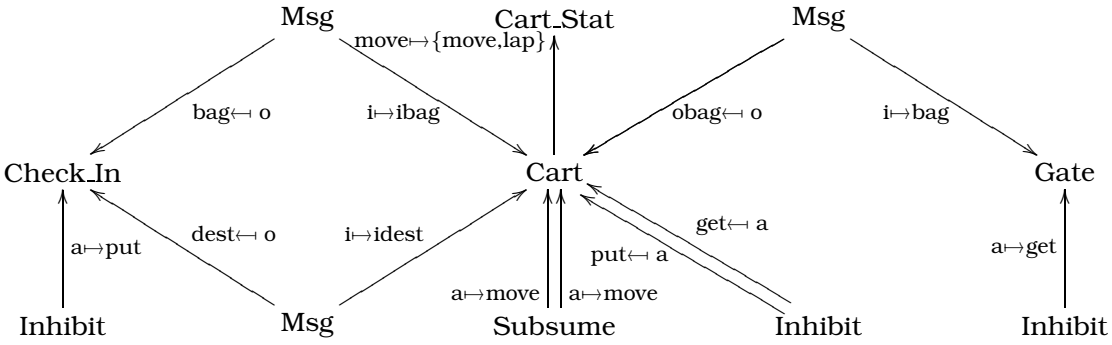
*Proof.* Similar to Proposition 4.15 on the preceding page and using $\mathcal{IP}$. ✓

To simulate a transient connector we need two rules, one to introduce the connector, the other to remove it. To make the specification easier, it is convenient to generate the second rule automatically from the first one. Basically it is the inverse rule, but some care must be taken. First, the variables of the left-hand side must be included in the right-hand side to make the inversion well-defined. Second, the state in which a connector is added is generally not the same as the state in which it is removed due to the computations meanwhile performed by the glue. Since the conditions under which the connector is in effect depend only on the state of the interacting components, the rule to remove a connector must use fresh state variables for the glue.

**Definition 4.69.** Given a dynamic reconfiguration rule $\langle p : (L \xleftarrow{l} K \xrightarrow{r} R), mc\rangle$ (possibly conforming to some style $AS$) such that $Vars(L) = Vars(R)$, the *mirror rule* is $\langle p : (R' \xleftarrow{r} K \xrightarrow{l} L), \neg mc\rangle$ where $R'$ is obtained from $R$ by replacing, for every $\langle P, \epsilon\rangle$ in $R \setminus r(K)$, each $\epsilon(o)$ by a different logical variable not in $Vars(L)$, if that does not violate the conditions on program instance morphisms. □

We now start presenting the reconfiguration rules for our running example. At each point in time, each cart is connected at most to one gate or one check-in counter. Therefore there is no need for message buses and we use only the simpler message passing connector. However, the message bus guarantees that the components' actions to send and receive messages are blocked when no interaction is occurring. Hence we must explicitly inhibit those actions when the message passing connector is not in effect. A further simplification is obtained by using a style instead of connectors with refined roles.

*Example 4.39.* The black-box view of the architectural style for the dynamic reconfiguration of the luggage distribution system is

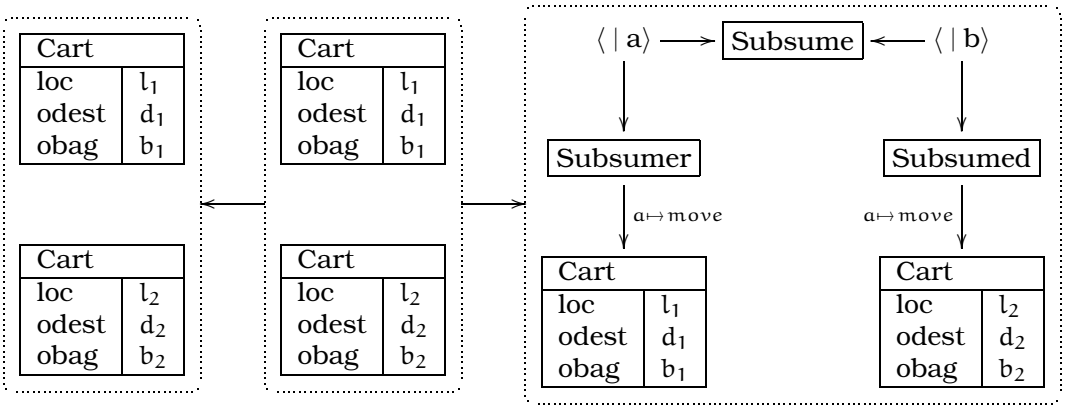



□

Notice that the above style, unlike the one in Example 4.34 on page 94, can be used for architectures with any number of components.

**Notation 4.70.** Due to page width constraints, we may omit the interface graph. A rule $L \xleftarrow{l} K \xrightarrow{r} R$ is simply written as $L \to R$ where the arrow is only used as a separator. It does not correspond to any total graph morphism. Also, a dynamic rule $\langle p, mc\rangle$ is written $p$ **if** $mc$ or simply $p$, if $mc$ is a tautology. □

*Example 4.40.* The rule to avoid a cart colliding with the one in front of it is:

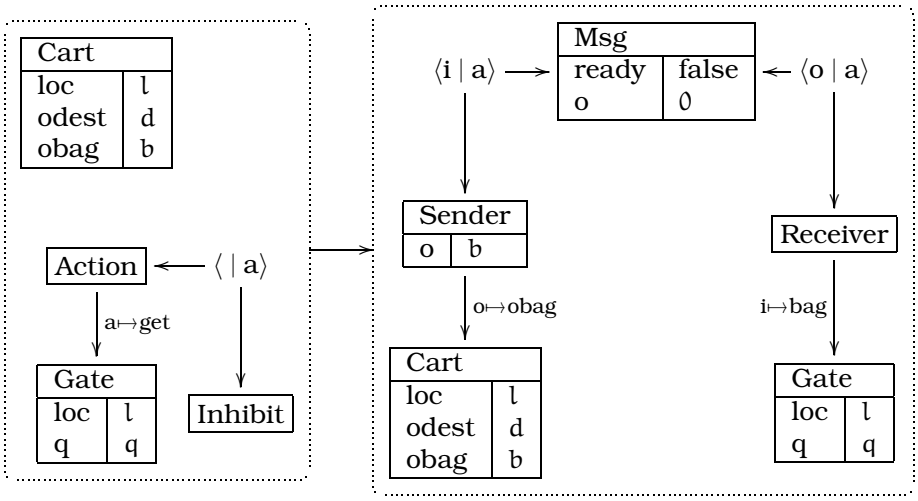



**if** $l_2 = l_1 +_N 1 \vee l_2 = l_1 +_N 2$

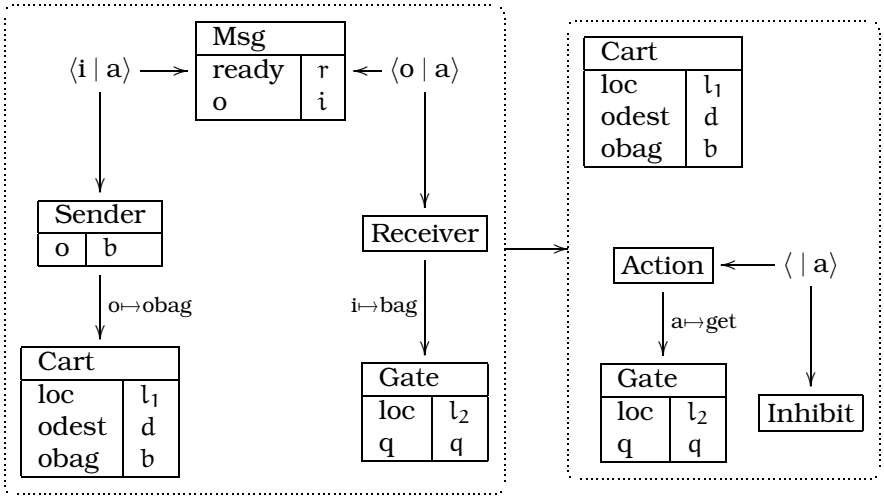
The mirror rule removes the action subsumption connector when it is not longer needed. □

As in the example above, where the added subsumption connector has no initialisation conditions in the glue and roles, it is often possible to prove that a rule will introduce program instances in a valid initial state for any substitution $\phi$. Thus the run-time check for each reconfiguration step becomes unnecessary, leading to a more efficient implementation.

*Example 4.41.* The next rule connects a cart to a gate when it passes in front of it. For illustration purposes, instead of using a matching condition we share the same logical variable to express co-location of cart and gate.




It should also be noticed that the rule does not check whether the cart is carrying a bag and, if so, whether the gate is the cart's destination. There are two possible scenarios: either the cart is carrying a bag for this gate or not. In the first case, the guard of the 'put' action is true and right after executing it, it becomes false. In the second case the guard is already false (see Example 4.3 on page 61) and the connector is not used at all. Now only action 'move' can execute (because 'get' is inhibited). This will change the cart's location and trigger the following rule to remove the connector.

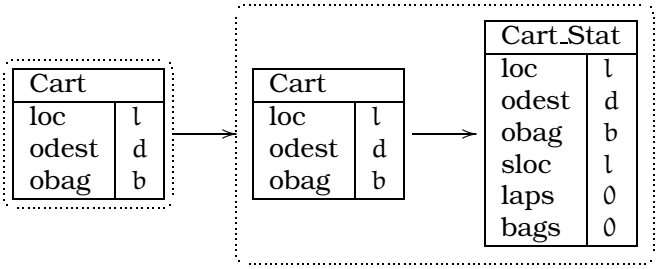



**if** $l_1 \neq l_2$ □

*Example 4.42.* A cart and a check-in station interact when they are co-located, the cart is empty, and the check-in has undelivered bags. In that case the cart gets a new bag and its destination (Figure 4.1 on the following page).

In the mirror rule the constants of the connector glues become new variables but the logical variables associated to the output variables of the 'Sender' roles cannot be changed due to the morphisms to 'Check_In' (Figure 4.2 on page 103). □

*Example 4.43.* The rule to refine a cart by one with statistics is

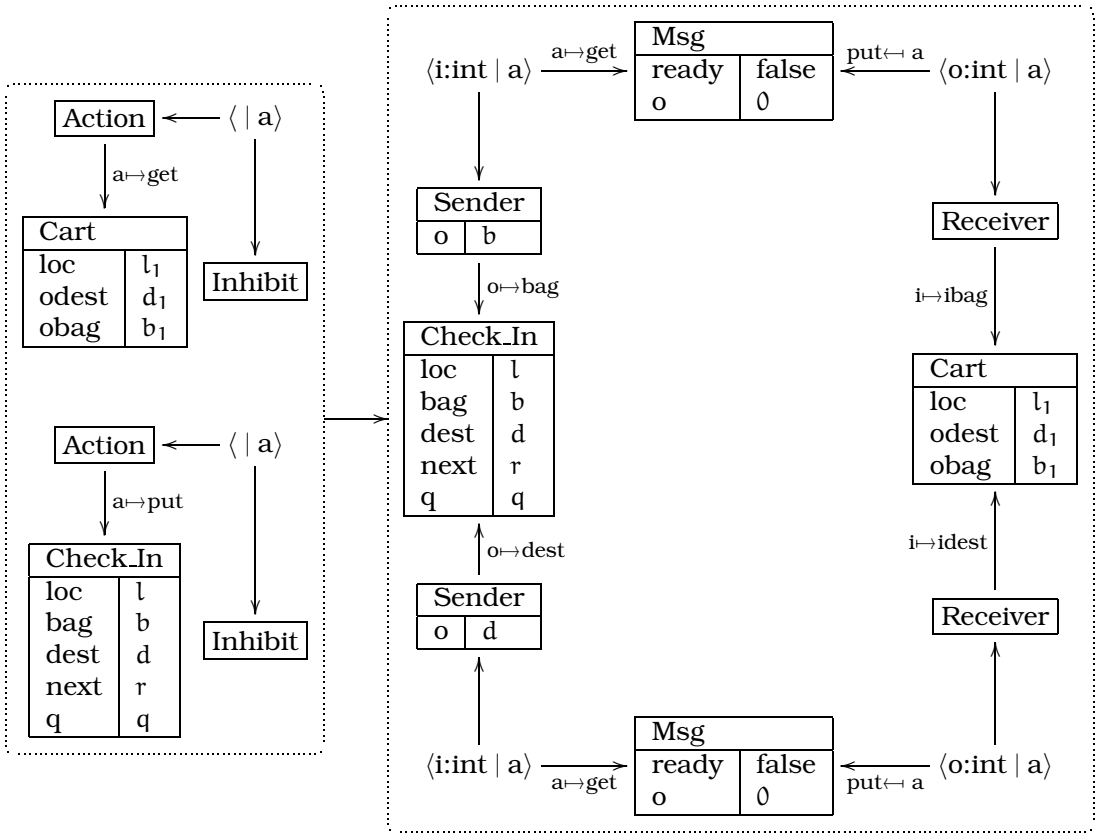



The definition of dynamic reconfiguration step implies $d = -1$ and $b = 0$. Putting it explicitly in the rule would be better because it would allow to check at "compile-time" that the initialisation condition is satisfied for any locations $l$. □

Now we turn to the "Oops, I forgot"-kind of reconfiguration. In this case the programmer (me) forgot to provide code to process the data gathered in the bag and lap counters[1]. Feeling that the problem is also due to the vague requirements provided by the project manager (myself), the programmer goes to the manager and, after a brief meeting, they decide to stop collecting further data after 100 laps and to collect the averages in a component specifically for that purpose.

*Example 4.44.* To achieve the goal we first write a "program" that has true initialisation, no actions, and simply holds the values for the accumulated bag and lap counters, together with the average. We also need two rules. The first creates the accumulator for a cart that has already made 100 laps. The second rule merges accumulators. When

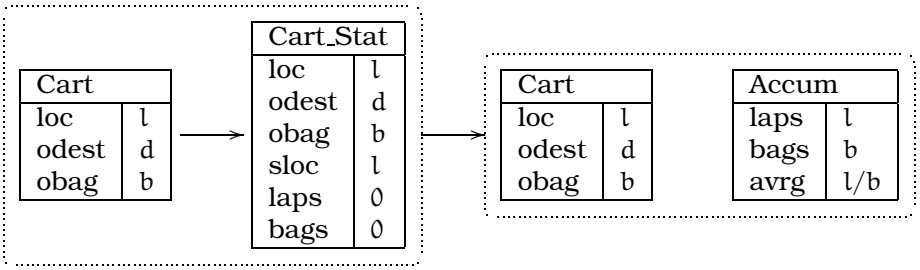
[1] This actually happened while I was writing this chapter.



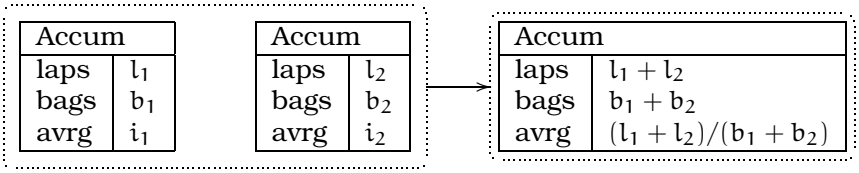

**if** $l = l_1 \wedge d_1 = -1 \wedge b_1 = 0 \wedge q \neq [\,]$

Figure 4.1: Before loading a bag from a check-in station

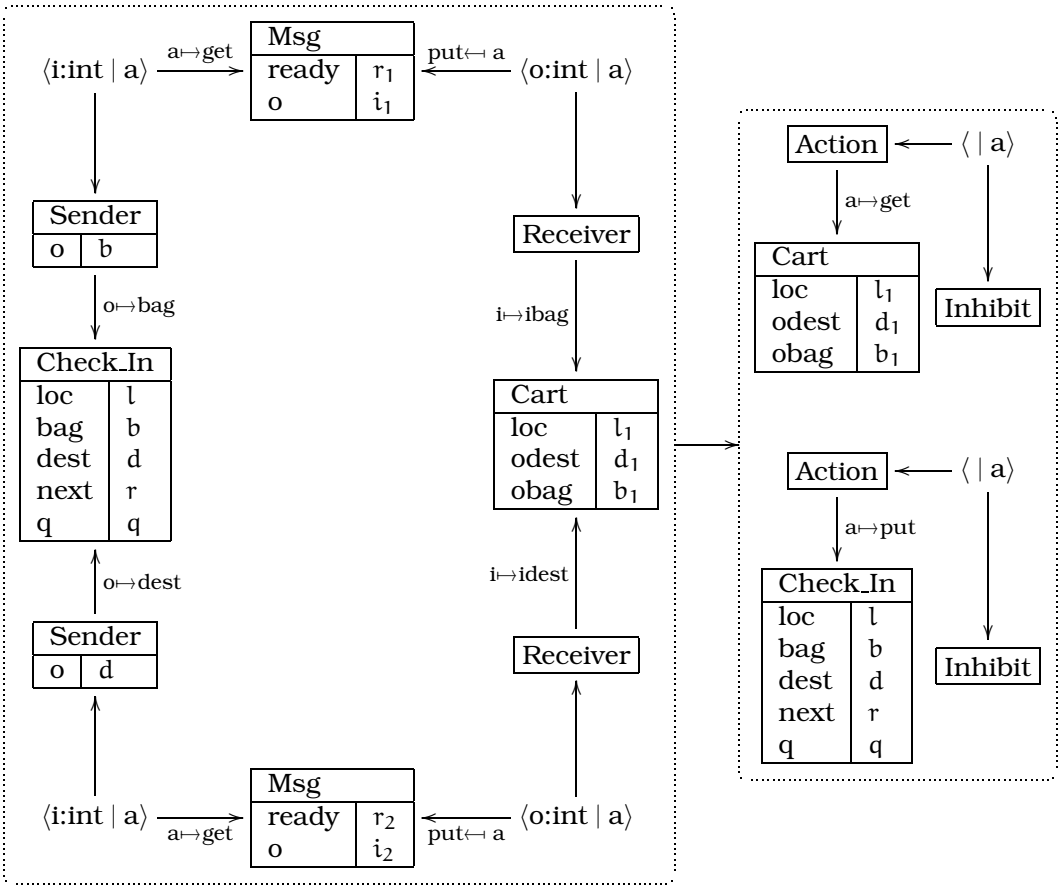
these rules can no longer be applied we obtain the final result.




**if** $l \geq 100$




It is obvious from the rules that the 'Accum' program cannot impose any conditions on the initial values of the two counters (besides being non-negative) and hence has a




**if** $l \neq l_1 \vee d_1 \neq -1 \vee b_1 \neq 0 \vee q = [\,]$

Figure 4.2: After loading a bag from a check-in station

true initialisation condition. □

Although it is not necessary for our example, the double-pushout approach guarantees that a 'Cart_Stat' is replaced by an accumulator only when it is not connected to any other component than 'Cart'. This is important both for conceptual reasons—components are not removed during interactions [KM90]—as technical ones: there will be no "dangling" roles.

This example shows how a rule describes transfer of state from an old to a new component. The transfer may involve both copy of values and arbitrarily complex calculations of new values from the old ones.

### 4.8.2 Process

An architecture instance is not just a labelled graph, it is a diagram with a precise semantics, given by its colimit. We can define a computation step of the system as being performed on the colimit and then propagated back to the components of the architecture through the inverse of their morphisms to the colimit. This keeps the state of the program instances in the architectural diagram consistent with the state of the colimit, and ensures that at each point in time the correct conditional rules are applied. As [MR98, HIM99] we adopt a two-phase approach: each computation step is

followed by a reconfiguration sequence. In this way, the specification of the components is simpler, because it is guaranteed that the necessary interconnections are in place as soon as required by the state of the components. In our example, a cart simply moves forward without any concern for its location. Without the guarantee that an action subsumption connector will exist whenever necessary, a cart would have to know at all times the locations of the other carts to be sure it would not collide with one of them. And this would make the program much more complex.

**Definition 4.71.** Given a style $AS$, an initial architecture instance $G$ conforming to $AS$, and a set of $AS$-dynamic reconfiguration rules, the configuration manager performs the following steps:

1. allow the user to add new connectors and components to $AS$;
2. allow the user to change the set of rules;
3. find a maximal sequence of $AS$-dynamic reconfiguration steps starting with $G$, obtaining a new diagram $G'$;
4. compute the colimit $S$ of $G'$;
5. if none of $S$'s actions can be executed, stop, otherwise update $S$'s environment according to the chosen action;
6. propagate back the changes to the environments of the program instances of $G'$, call the new diagram $G$, and go back to step 1. □

The first two steps cater for ad-hoc reconfiguration. For our system, step 1 allows to add the 'Accum' program to the style of Example 4.39 on page 100, and step 2 permits the addition of the rules in Example 4.44 on the page before.

## 4.9 Concluding Remarks

This chapter presents an algebraic foundation for software architecture reconfiguration. The approach is based on three pillars: the general framework of Category Theory; the category of typed graphs and their morphisms; the category of CommUnity programs with morphisms that capture superposition and refinement. The first two allow us to use in a straightforward way the double pushout approach to graph transformation.

The main advantages of this approach are:

- Architectures, reconfigurations, and connectors are represented and manipulated in a graphical yet mathematical rigorous way at the same language-independent level of abstraction, resulting in a very uniform framework based simply on diagrams and their colimits.
- The chosen program design language is at a higher level of abstraction than process calculi or term rewriting, allowing a more intuitive representation of program state and computations.
- Computations and reconfigurations are kept separate but related in an explicit, simple, and direct way through the colimit construction.
- Typed graph morphisms provide a simple, declarative, yet expressive notion of architectural style.
- Several practical problems—maintaining the style during reconfiguration, transferring the state during replacement, removing components in a quiescent state, adding components properly initialized—are easily handled.

Another advantage is transient connectors to encapsulate, localise, and make changing interactions explicit. This contrast with other approaches [MR98, CFM98]. It also reinforces the importance of connectors to support reconfiguration, as other researchers have already observed [OT98].

Among the many possibilities for future work, we are considering the following:

- Implement the approach, either by incorporating a library to compute colimits [Wol98] into a CommUnity tool that is being developed, or by adapting a graph transformation workbench to CommUnity.
- Extend the work done to the full language [Lop99].
- Look into and try to adapt work on graph rewriting termination [Plu95] and sequential independence to be able to analyse the possible reconfiguration sequences.
- Investigate further into primitive connectors, connector operations, and their algebraic properties along the lines of [Gar98].
- Investigate the applicability of the approach to other languages. This should only require changing the language-dependent notions of program instance, superposition/refinement, channel, and data view, because the remaining definitions (connector, architecture style, dynamic reconfiguration rule, etc.) build on them.
- Handle hierarchic architectures, possibly representing a composite component as the colimit of a sub-architecture.
- Explore the expressiveness of Category Theory to represent multiple views and make their relationships explicit (e.g., through functors).

# Chapter 5

# Conclusion

We surveyed work done in Software Engineering, Distributed Systems, Mobile Computing, and Theoretical Computer Science in order to investigate several aspects of the formal specification of dynamic reconfiguration of software architectures.

We looked for existing mathematical constructions and formalisms to avoid the "yet another language/formalism" syndrome. In our case we chose graphs, Category Theory, the chemical computation model, and UNITY, all of them widely used in several subareas of Computer Science.

We presented three approaches to the formal specification of software architecture reconfiguration. They make different assumptions on the systems to be applied to (e.g., whether they are hierarchic or not), and they emphasize different aspects, namely efficiency, simplicity, and abstraction.

The transaction approach addresses the minimization of the disruption caused by reconfiguration. Given a hierarchic architecture and the dependencies between the transactions among the various components, together with a a pre-defined, fixed, and valid set of addition and deletion commands, the approach generates the specification of the architecture of a configuration manager that executes those commands in a correct, modular, and minimal way. The input and output being architectures, it follows that the main advantage of the approach is that managers are specified in exactly the same way as the systems on which they operate. Additionally, the decomposition of the configuration manager mirrors that of the system.

The CHAM approach strives for maximal uniformity and minimal number of constructions used by the specification formalism. Architectures (as members of a certain style), reconfigurations, and computations are all specified by rewriting of multisets of terms. The main contribution is a systematic way to write individual component specifications—each describing the component's structure (which may be a subarchitecture), state, interactions, computations, and allowed reconfigurations—and to compose them into architectures.

The COMMUNITY approach, on the one hand, provides an abstract algebraic framework to describe and operate on connectors, styles, and architectures and, on the other hand, raises the level of component behaviour description. As in the CHAM approach, reconfigurations are described by conditional graph rewriting. The conditions state under which conditions a change may occur, and rewriting allows the designer to capture higher level abstractions than those provided by the transaction approach, like component relinking and replacement. Unlike the CHAM approach however, it is able to automatically enforce the maintenance of a given style during reconfiguration. The major strength of this approach is to avoid the drawbacks listed at the end of Section 1.4 on page 3.

Although the approaches are quite disparate, they have the following in common:

- all four basic changes—addition and removal of components and connections—are allowed;
- architectures are represented explicitly through the simple concept of graph.

The transaction approach only deals with the execution of the reconfiguration commands, describing it by a partial order which is represented as a graph. The other two approaches specify reconfigurations through conditional graph rewrite rules. The partial order graph and the rules share the following characteristics:

- they are suitable for their purposes and for the underlying representation of architectures;
- they are conceptually simple, and their pictorial representation makes them easy to understand;
- they represent the reconfiguration process explicitly, either showing the temporal ordering of the commands or the parts of the architecture to be modified.

Taking all this into account, we conclude that our goal, stated in Section 1.5 on page 6, has been achieved, hence improving on the previous work summarised in Section 1.4 on page 3: arbitrary software architecture reconfigurations can be formally specified in a simple, explicit, and adequate way.

Future work to augment our contributions will be along three axes: analysis, tools, and integration.

The purpose of reconfiguration is to obtain an architecture with new properties while preserving of the existing architecture (like absence of deadlock). It is desirable to describe those properties in a precise and declarative way in order to check them against the reconfiguration specification. The description language and the analysis techniques may depend on the kind of properties (e.g., security, throughput). It is also necessary to analyse whether a reconfiguration script performs the intended changes and terminates.

Tools will provide the necessary support both for specification and analysis. The goal is to obtain an integrated graphical environment for reconfigurable software architectures that allows to describe components, interconnect them into architectures, manage component and architecture libraries, specify reconfigurations, animate them, and automatically validate some properties.

Integration will be done in two "directions". The "vertical" integration will try to combine the transaction approach with each one of the other two. The goal is to derive from the CHAM (or CommUnity) description the exact reconfiguration script to be executed by the configuration manager, in order to have an integrated framework that provides the specification and execution of architectural changes. This will require several changes to the work done because the assumptions made by the transaction approach—dependent transactions between components with unknown state—are different from the CHAM and CommUnity approaches.

The "horizontal" integration aims at incorporating in the CommUnity approach some of the features of the CHAM approach (e.g., hierarchic architectures and self-organisation) and vice-versa (e.g., connectors). The goal is to allow both approaches to handle the same kinds of architectures and reconfigurations, although specifying them in different ways. The user would thus choose between the two approaches according to other criteria, e.g., the way computations and interactions are described. The purpose of this kind of integration is not to obtain a single framework, because that would defeat the rationale of using different formalisms to emphasize different aspects and handle different problems.

# Appendix A

# Mathematics

This appendix provides the basic mathematical definitions and notations used in Chapter 4 on page 49. To facilitate exposition and achieve greater uniformity we base most definitions on the familiar concept of graph. We do not prove well-known results. For notation used but not defined in this appendix see the List of Symbols on page viii.

## A.1 Graphs

Although the concepts are familiar, there is not one standard terminology and notation for graphs, hence the need for this section.

**Definition A.1.** A *graph* is a tuple $\langle N, A, src, trg \rangle$ where

- $N$ is a collection of *nodes*,
- $A$ is a collection of *arcs*,
- $src, trg : A \to N$ map each arc to its *source* and *target* node, respectively. □

**Notation A.2.** An arc $a$ with source $x$ and target $y$ is written $x \xrightarrow{a} y$. □

**Definition A.3.** Given a graph with nodes $x$ and $y$, a *path* of *length* $n > 0$ from $x$ to $y$ is a sequence of arcs $a_1, a_2, \ldots, a_n$ such that $src(a_1) = x$, $trg(a_n) = y$, and $trg(a_i) = src(a_{i+1})$ for $0 < i < n$. A path of length zero is the empty sequence and is defined only for $x = y$. □

**Notation A.4.** For a given graph $G$, the collection of paths of length $i$ is written $G_i$. □

The collection $G_0$ corresponds to the graph's nodes, and $G_1 = A$.

**Definition A.5.** The *outdegree* (resp. *indegree*) of node $x$ is the number of arcs $a$ such that $src(a) = x$ (resp. $trg(a) = x$). □

A graph morphism is a structure preserving mapping of nodes and arcs.

**Definition A.6.** A *graph morphism* $f : G \to G'$ is a pair of functions $f_N : N \to N'$ and $f_A : A \to A'$ such that $trg; f_N = f_A; trg'$ and $src; f_N = f_A; src'$. □

**Notation A.7.** To be able to present several graph morphisms within the same picture, we write graphs within dotted boxes, and morphisms are arcs between graphs showing the mappings done by $f_N$ and $f_A$ (see Example A.1 on the next page). □

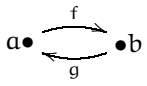
We adopt from [CMR96b] the concept of a graph G typed over a fixed graph of types TG. The purpose of TG is to restrict the possible nodes and what arcs are allowed between a pair of nodes. The typing of G is provided by a morphism to TG. A morphism between typed graphs must preserve the typing.

**Definition A.8.** A *typed graph* (over a *graph of types* TG) is a pair $\langle G, t\rangle$ where $t : G \to TG$ is a graph morphism. A *typed graph morphism* $m : \langle G, t\rangle \to \langle G', t'\rangle$ is a graph morphism $m : G \to G'$ such that $m; t' = t$. □

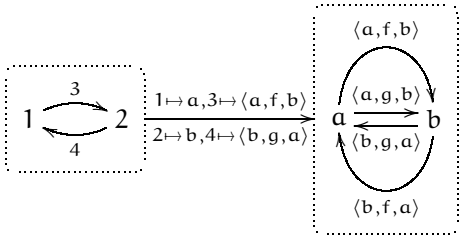
A special case of typed graphs are labelled graphs: TG contains one node for each node label, and between each pair of nodes there is one arc for each arc label.

**Definition A.9.** A *labelled graph* over the collections of *node labels* $L_N$ and *arc labels* $L_A$ is a graph $\langle G, lbl\rangle$ typed over $TG = \langle L_N, L_N \times L_A \times L_N, \pi_1, \pi_3\rangle$. □

*Example A.1.* Let $L_N = \{a, b\}$ and $L_A = \{f, g\}$. Then the labelled graph




is the typed graph

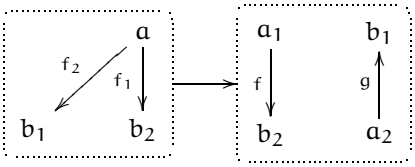



□

From the definition of typed graph morphism, it results that a labelled graph morphism must preserve the labels on nodes and arcs. We use the following conventions.

**Notation A.10.** Nodes (or arcs) labelled with the same label are distinguished through numeric indices. A mapping is omitted when it can be unambiguously determined from the labels and indices used. □

*Example A.2.* Using the same sets of labels as in the previous example,




is well-defined since there is only one morphism that preserves structure and labels: $\{a \mapsto a_1, f_1 \mapsto f, f_2 \mapsto f, b_1 \mapsto b_2, b_2 \mapsto b_2\}$. □

A labelled graph with a distinguished node is called a labelled transition system.

**Definition A.11.** A *labelled transition system* is a labelled graph, whose nodes are called *worlds* and whose arcs are called *transitions*, with a distinguished node called *initial world*. □

**Notation A.12.** We use the symbols $W$, T, and $w_0$ to represent the collections of worlds and transitions, and the initial world, respectively. □

## A.2 Category Theory

Category Theory [Pei91, Mar96, FM94] is the mathematical discipline that studies, in a general and abstract way, relationships between arbitrary entities. Basically, a category is a graph, with nodes called objects and arcs called morphisms, such that paths are closed under transitivity and reflexivity.

**Definition A.13.** A *category* $\mathfrak{C}$ is a tuple $\langle G_{\mathfrak{C}}, ;, id\rangle$ where

1. $G_{\mathfrak{C}} = \langle |\mathfrak{C}|, Hom_{\mathfrak{C}}, dom, cod\rangle$ is a graph, with *objects* $|\mathfrak{C}|$, *morphisms* $Hom_{\mathfrak{C}}$, *domain* map $dom$, and *codomain* map $cod$,
2. $; : (G_{\mathfrak{C}})_2 \to (G_{\mathfrak{C}})_1$ is the *composition* operator,
3. $id : |\mathfrak{C}| \to Hom_{\mathfrak{C}}$ maps each object to its *identity morphism*,

such that

- $dom(f;g) = dom(f)$ and $cod(f;g) = cod(g)$,
- $f;(g;h) = (f;g);h$,
- $dom(id(x)) = cod(id(x)) = x$,
- $f;id(cod(f)) = id(trg(f));f = f$. □

**Notation A.14.** The collection of all morphisms with domain $x$ and codomain $y$ is written $Hom_{\mathfrak{C}}(x,y)$. A morphism $f \in Hom_{\mathfrak{C}}(x,y)$ is written $f : x \to y$ or $x \xrightarrow{f} y$. □

*Example A.3.* The category $\mathfrak{Set}$ has sets as objects and total functions between sets as morphisms. Composition of morphisms is given by the usual composition of total functions and the identity morphisms correspond to identity functions. □

In this case, between each pair of sets there are as many morphisms as total functions between those sets. Other morphisms are possible. For instance, the subset relation. In this case there is at most one morphism between a pair of sets, and the composition operator just states that the relation is transitive.

*Example A.4.* Graphs and graph morphisms form the category $\mathfrak{Graph}$. □

It is also possible to have "morphisms" between different categories. Such morphisms must preserve compositions and identities.

**Definition A.15.** Given categories $\mathfrak{C}$ and $\mathfrak{C}'$, a *functor* $\mathcal{F} : \mathfrak{C} \to \mathfrak{C}'$ is a graph morphism $\mathcal{F} : G_{\mathfrak{C}} \to G_{\mathfrak{C}'}$ such that

- $\forall x \in |\mathfrak{C}|\ \mathcal{F}_A(id(x)) = id'(\mathcal{F}_N(x))$,
- $\forall f_1 f_2 \in (G_{\mathfrak{C}})_2\ \mathcal{F}_A(f_1;f_2) = \mathcal{F}_A(f_1);'\mathcal{F}_A(f_2)$.

The functor is called *forgetful* if $\mathfrak{C}$ has more structure than $\mathfrak{C}'$. □

Given a category, it is possible to build another one by distinguishing one of its elements.

**Definition A.16.** Given a category $\mathfrak{C}$ with an object $x$, the *comma category* $(\mathfrak{C} \downarrow x)$ has as objects all pairs $\langle y, f\rangle$ with $f \in Hom_{\mathfrak{C}}(y,x)$ and the morphisms $g : \langle y_1, f_1\rangle \to \langle y_2, f_2\rangle$ are all $g \in Hom_{\mathfrak{C}}(y_1, y_2)$ such that $g;f_2 = f_1$. □

*Example A.5.* The category $\mathfrak{Graph}_{TG}$ of graphs typed over $TG$ is the comma category $(\mathfrak{Graph} \downarrow TG)$. □

The inverse construction is trivial and amounts to forget the extra morphisms to $x$.

**Proposition A.1.** *For any* $(\mathfrak{C} \downarrow x)$ *there is a forgetful functor* $\mathcal{F}_x : (\mathfrak{C} \downarrow x) \to \mathfrak{C}$. □

Diagrams are labelled graphs—where nodes denote objects and arcs represent morphisms—and can be used to represent "complex" objects as configurations of smaller ones. The labelling of the nodes and arcs must of course be consistent with the objects and morphisms provided by the category.

**Definition A.17.** A *diagram* in category $\mathfrak{C}$ is a graph typed over $G_{\mathfrak{C}}$. □

**Notation A.18.** We use $D = \langle \Delta, \delta \rangle$ or simply $\langle \Delta_D, \delta_D \rangle$ to denote a diagram $D$ and depict it with a labelled graph, where the label of node (or arc) $x$ is $\delta(x)$. □

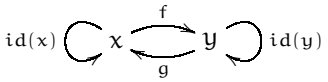
Functors allow us to "translate" diagrams from one category to another.

**Notation A.19.** If $D = \langle \Delta, \delta \rangle$ is a diagram in $\mathfrak{C}$, and $\mathcal{F} : \mathfrak{C} \to \mathfrak{C}'$ is a functor, then $\mathcal{F}(D)$ is the diagram $\langle \Delta, \delta; \mathcal{F} \rangle$ in $\mathfrak{C}'$. □

A diagram commutes if all paths between the same pair of nodes represent the same morphism.

**Definition A.20.** A diagram $\langle \Delta, \delta \rangle$ in category $\mathfrak{C}$ is *commutative* if for every pair $x, y \in N$ and every pair of paths $a_1 \dots a_n$ and $a'_1 \dots a'_m$ from $x$ to $y$, $\delta(a_1); \dots; \delta(a_n) = \delta(a'_1); \dots; \delta(a'_m)$ in $\mathfrak{C}$. □

Two objects $x$ and $y$ are said to be *isomorphic* if there is a commutative diagram

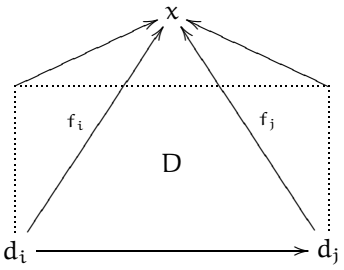



**Definition A.21.** A morphism $f : x \to y$ is an *isomorphism*, and $x$ and $y$ are *isomorphic*, if there is a morphism $g : y \to x$ such that $f; g = id(x)$ and $g; f = id(y)$. □

*Example A.6.* In $\mathfrak{Set}$, isomorphisms are bijective functions. □

For certain categories, each diagram denotes an object that can be retrieved through an operation called colimit. Informally, the colimit of a diagram returns the "minimal" object such that there is a morphism from every object in the diagram to it (i.e., the colimit contains the objects in the diagram as components) and the addition of these morphisms to the original configuration results in a commutative diagram (i.e., interconnections, as established by the morphisms of the configuration diagram, are enforced).

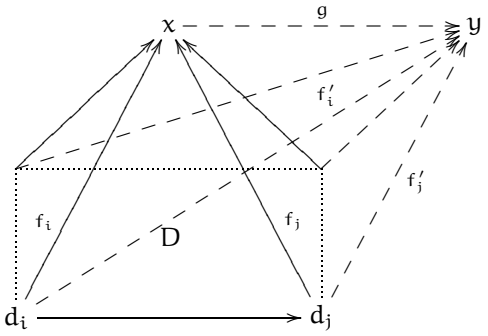
The mathematical definition is given in two steps. First we define a cocone for a diagram $D$ as a commutative diagram of the form



**Definition A.22.** A *cocone* for a diagram $\langle \Delta, \delta \rangle$ in category $\mathfrak{C}$ is a *cocone object* $x \in |\mathfrak{C}|$ and one morphism $f_i \in Hom_{\mathfrak{C}}(\delta(n_i), x)$ for each node $n_i$ of $\Delta$, such that for every arc $n_i \xrightarrow{a_{ij}} n_j$ of $\Delta$, $\delta(a_{ij}); f_j = f_i$. □

**Notation A.23.** A cocone is written $\{f_i : d_i \to x\}$, where $d_i = \delta(n_i)$. □

The second step is to define colimit as the minimal cocone:




**Definition A.24.** A *colimit* for diagram $D$ in category $\mathfrak{C}$ is a cocone $\{f_i : d_i \to x\}$ such that if $\{f'_i : d_i \to y\}$ is another cocone for $D$, then there is a unique morphism $g : x \to y$ such that $f_i; g = f'_i$. Object $x$ is called the *colimit object*. □

Due to the next result, we usually refer to *the* colimit of a given diagram.

**Proposition A.2.** *All colimit objects for the same diagram are isomorphic.* □

The colimits of particular diagrams have special names.

**Definition A.25.** A colimit for the empty diagram is called an *initial object*. □

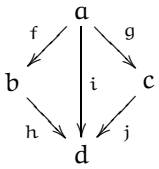
Any object is a cocone for the empty diagram. The colimit is then an object that has a unique morphism to any other object.

*Example A.7.* The initial object of $\mathfrak{Set}$ is the empty set. □

**Definition A.26.** The colimit of a diagram with just two objects is called *coproduct*. □

*Example A.8.* In $\mathfrak{Set}$, the coproduct is the disjoint union. □

Pushouts are colimits of diagrams of the form $b \xleftarrow{f} a \xrightarrow{g} c$. By definition of colimit, the pushout returns an object $d$ such that the diagram




exists and commutes (i.e., $f; h = i = g; j$). Furthermore, for any other pushout candidate $d'$, there is a unique morphism $k : d \to d'$. This ensures that $d$, being a component of any other object in the same conditions, is minimal. Morphisms $g$ and $j$ are called the pushout complement of $f$ and $h$, and vice-versa.

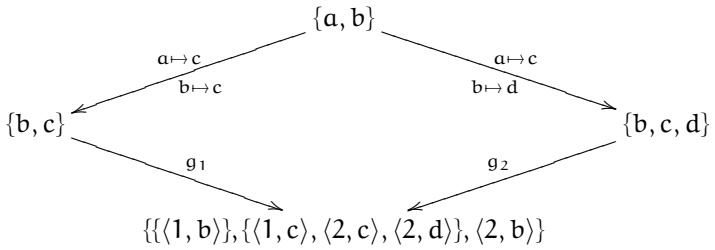
**Definition A.27.** A *pushout* is a colimit for a diagram of the form $x_1 \xleftarrow{f_1} x \xrightarrow{f_2} x_2$. □

**Definition A.28.** A *pushout complement* of $x_0 \xrightarrow{f_1} x_1 \xrightarrow{g_1} x$ is $x_0 \xrightarrow{f_2} x_2 \xrightarrow{g_2} x$ such that $\{g_i : x_i \to x\}$ is a pushout of $\{f_i : x_0 \to x_i\}$, with $i = 1, 2$. □

The pushout of sets is obtained by computing first the coproduct (i.e., the disjoint union) and then calculating the equivalence classes of those elements identified through $x$.

*Example A.9.* In $\mathfrak{Set}$, the pushout $\{g_i : x_i \to z\}$ of $x \xrightarrow{f_i} x_i$ (with $i = 1, 2$) is given by $z = \{\langle i, e_i\rangle \mid e_i \in x_i\}/\equiv$ and $g_i(e_i) = [\langle i, e_i\rangle]_\equiv$, with $\equiv$ the equivalence relation obtained from $\langle 1, e_1\rangle \sim \langle 2, e_2\rangle \iff \exists e \in x\ f_1(e) = e_1 \wedge f_2(e) = e_2$.

For instance,




with $g_1(b) = \{\langle 1, b\rangle\}$, $g_1(c) = g_2(c) = g_2(d) = \{\langle 1, c\rangle, \langle 2, c\rangle, \langle 2, d\rangle\}$, and $g_2(b) = \{\langle 2, b\rangle\}$. □

This example clearly shows that Category Theory enforces a locality principle in the sense that the names used to write down objects are independent of each other. The relationships are solely established by the morphisms. In this case, the name $b$ used in the different sets does not represent the same element as the mappings clearly show. In fact, sets with the same cardinality are isomorphic and thus indistinguishable, since they are equipped with the same morphisms. Therefore, the names we choose to represent the elements of sets are irrelevant.

As a further example that the relevant information lies in the morphisms, not in the internal structure of the objects, consider the category mentioned after Example A.3 on page 111: objects are sets and morphisms show the "subset-of" relation. The colimit of a diagram in that category is the minimal superset (i.e., the union) of the sets appearing in the diagram. Notice that although the objects are the same as in $\mathfrak{Set}$, the change in the definition of morphism brought about a change in the meaning of the colimit.

We are interested in categories that provide a "semantics" for every diagram.

**Definition A.29.** A category $\mathfrak{C}$ is *finitely cocomplete* if every finite diagram in $\mathfrak{C}$ has a colimit. □

**Theorem A.3.** *A category is finitely cocomplete if and only if it has an initial object and a pushout for every pair of morphisms with common domain.* □

*Example A.10.* $\mathfrak{Set}$ is finitely cocomplete. □

Computing the colimit in comma categories amounts to calculate it in the underlying category, and the same for pushout complements.

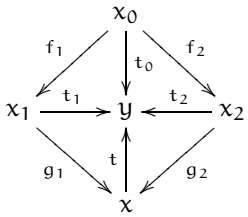
**Proposition A.4.** *If $\mathfrak{C}$ is finitely cocomplete, so is any $(\mathfrak{C} \downarrow y)$.* □

*Proof.* Let $D$ be any finite diagram in $(\mathfrak{C} \downarrow y)$ and $\{f_i : x_i \to x\}$ be the colimit of $\mathcal{F}_y(D)$. Then $\{f_i : \langle x_i, t_i\rangle \to \langle x, t\rangle\}$ is the colimit of $D$ with $t : x \to y$. Since $\{t_i : x_i \to x\}$ is a cocone of $\mathcal{F}(D)$, $f_i$ are morphisms in $(\mathfrak{C} \downarrow y)$ and $t$ exists and is unique.

It remains to show that for any cocone $\{f_i : \langle x_i, t_i\rangle \to \langle x', t'\rangle\}$ there is a unique $f : \langle x, t\rangle \to \langle x', t'\rangle$ such that for every $i$, $f_i; f = f'_i$. Because $\{f'_i : x_i \to x\}$ is a cocone of $\mathcal{F}_y(D)$, $f$ satisfies the conditions. It suffices to prove $f \in \mathrm{Hom}_{(\mathfrak{C}\downarrow y)}$. Let $\overline{t} = f; t$. Then $f_i; \overline{t} = f_i; f; t = f'_i; t = t_i$. Since $x$ is a pushout object, $t$ is the unique morphism satisfying $f_i; t = t_i$ and therefore $\overline{t} = t$ as wished. ✓

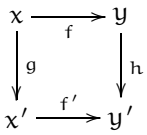
**Proposition A.5.** *A diagram* $\langle x_0, t_0\rangle \xrightarrow{f_1} \langle x_1, t_1\rangle \xrightarrow{g_1} \langle x, t\rangle$ *in* $(\mathfrak{C} \downarrow y)$ *has pushout complement if and only if diagram* $x_0 \xrightarrow{f_1} x_1 \xrightarrow{g_1} x$ *in* $\mathfrak{C}$ *has.* □

*Proof.* The left to right implication is immediate using $\mathcal{F}_y$. For the other direction, consider the following diagram in $\mathfrak{C}$:




If $f_2$ and $g_2$ are a pushout complement of $f_1$ and $g_1$ in $\mathfrak{C}$, they are also in $(\mathfrak{C} \downarrow y)$ with $t_2 = g_2; t$. They are morphisms in $(\mathfrak{C} \downarrow y)$ because $f_2; t_2 = f_2; g_2; t = f_1; g_1; t = f_2; t_1 = t_0$. ✓

Finally, we introduce a general operation on diagrams that replaces every labelled arc $x \xrightarrow{f} y$ by $x' \xrightarrow{f'} y'$ if there are morphisms $g : x \to x'$ and $h : y \to y'$ such that




commutes.

**Definition A.30.** Diagram $\langle \Delta', \delta'\rangle$ *specialises* diagram $\langle \Delta, \delta\rangle$ in the same category if $\Delta' = \Delta$ and there is a collection $\{f_n \in \mathrm{Hom}_{\mathfrak{C}}(\delta(n), \delta'(n)) \mid n \in N\}$ such that $\forall a \in A\ \delta(a); f_{trg(a)} = f_{src(a)}; \delta'(a)$. □

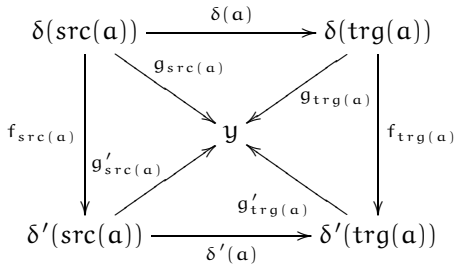
This operation preserves the semantics of the original diagram in the following sense.

**Proposition A.6.** *If* D *is specialised by* D′, *x is a colimit object of* D, *and y is a colimit object of* D′, *then there exists a morphism with domain x and codomain y.* □

*Proof.* The result is immediate from Definition A.22 on page 113 if we prove that $y$ is a cocone object of D. Let $\{g'_n : \delta'(n) \to y\}$ be a colimit of D′. We prove $\{g_n : \delta(n) \to y\}$ is a cocone of D, with $g_n = f_n; g'_n$. Let $a \in A$. Then

$$\begin{aligned}
\delta(a); g_{trg(a)} &= \delta(a); f_{trg(a)}; g'_{trg(a)} && \text{definition of } g\\
&= f_{src(a)}; \delta'(a); g'_{trg(a)} && \text{Definition A.30}\\
&= f_{src(a)}; g'_{src(a)} && \text{colimit of D}'\\
&= g_{src(a)} && \text{definition of } g \quad ✓
\end{aligned}$$

The proof establishes that the following diagram commutes.



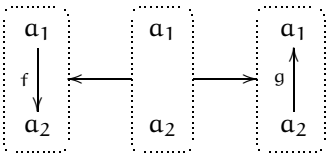
## A.3 Graph Grammars

The algebraic approach to graph transformation was introduced over 20 years ago in order to generalize grammars from strings to graphs. Hence it was necessary to adapt string concatenation to graphs. The approach is algebraic because the gluing of graphs is done by a pushout in an appropriate category. There are two main variants, the double-pushout approach [CMR+96a] and the single-pushout approach [EHK+96]. We first present the former which is based on $\mathfrak{Graph}$. We also take the opportunity to extend the definitions for $\mathfrak{Graph}_{TG}$ as done in [CMR96b].

A graph transformation rule, called graph production, is simply a diagram of the form $L \xleftarrow{l} K \xrightarrow{r} R$ stating how graph $L$ is transformed into $R$, where $K$ is the common subgraph, i.e., those nodes and arcs that are not deleted by the rule.

**Definition A.31.** A *graph production* $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ is composed of a *production name* $p$ and two injective graph morphisms $l : K \to L$ and $r : K \to R$, where $L$, $K$, and $R$ are called the *left-hand side*, the *interface*, and the *right-hand side* of $p$, respectively. A production is *typed over* $TG$ if $l$ and $r$ are morphisms in $\mathfrak{Graph}_{TG}$. □

*Example A.11.* Consider the graph of types of Example A.1 on page 110 and the conventions in Notation A.10 on page 110. The rule




substitutes an arc by another. □

A production can be applied to a graph $G$ if the left-hand side can be matched to $G$, i.e., if there is a graph morphism $m : L \to G$. The transformed graph is then obtained through two pushouts.

**Definition A.32.** Given a graph $G$, a production $p : (L \xleftarrow{l} K \xrightarrow{r} R)$, and a morphism $m : L \to G$, a *direct derivation* from $G$ to $H$ using $p$ based on *match* $m$, written $G \overset{p,m}{\Longrightarrow} H$, exists if diagram

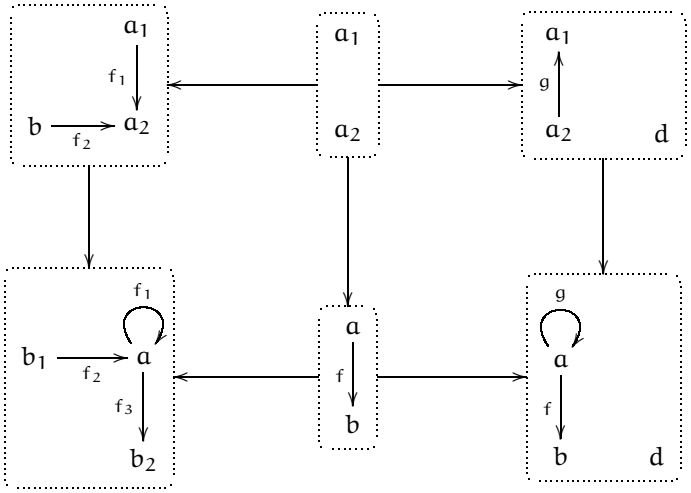
$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle d} & & \downarrow{\scriptstyle m^*} \\ G & \xleftarrow{l^*} & D & \xrightarrow{r^*} & H \end{array}$$

can be constructed, where each square is a pushout. Morphism $m^*$ is called the *co-match* of the derivation. A direct derivation is typed over $TG$ if the above is a diagram in $\mathfrak{Graph}_{TG}$. □

Intuitively, first the pushout complement object $D$ is obtained by deleting from $G$ all nodes and arcs that appear in $L$ but not in $K$. Then $H$ is obtained by adding to $D$ all nodes and arcs that appear in $R$ but not in $K$. For more details see [MEN96].

*Example A.12.* We extend the rule of Example A.11 to show, besides substitution of an arc, the removal of a connected node, and the creation of an unconnected node. Furthermore, the derivation is based on a non injective match. All morphisms are

uniquely determined.




□

A direct derivation is only possible if the pushout complement given by $d$ and $l^*$ exists. For that to happen, the match $m$ must obey two conditions. The dangling condition states that if the production removes a node $n \in L$, then each arc incident to $m(n) \in G$ must be image of some arc attached to $n$. The identification condition imposes that if the production removes one node (or arc) and maintains another one, then $m$ may not map them to the same node (or arc) in $G$.

*Example A.13.* The left diagram violates the dangling condition, the right one the identification condition, no matter which morphism $l$ is chosen.


□

Both conditions are quite intuitive. The first one prevents dangling arcs, the second one avoids contradictory situations. Both allow an unambiguous prediction of removals. A node of $G$ will be removed only if its context (i.e., adjacent arcs and nodes) are *completely* matched by the left-hand side of some production. The advantage is that the production specifier can control exactly in which contexts a node is to be deleted. This means it is not possible to remove a node no matter what other nodes are linked to it.

**Proposition A.7.** *Morphisms $f : G \to G'$ and $g : G' \to G''$ in $\mathfrak{Graph}$ (or $\mathfrak{Graph}_{TG}$) have a pushout complement if and only if the following two conditions are met:*

**dangling condition** *No arc in $A'' \setminus g_A(A')$ is incident to any node in $g_N(N' \setminus f_N(N))$;*

**identification condition** *There are no nodes or arcs $x, y \in G'$ such that $g(x) = g(y)$ and $y \notin f(G)$.*

*In this case we say $g$ satisfies the* gluing condition *w.r.t. $f$. If $f$ is injective then the pushout complement is unique up to isomorphism.* □

This explains why in a graph production the left-hand side morphism $l$ is injective. The right-hand side morphism is also required to be injective to allow derivations to be invertible, hence providing an "undo" facility.

**Proposition A.8.** *For each direct derivation* $G \overset{p,m}{\Longrightarrow} H$ *(possibly typed over* $TG$*) there is an* inverse derivation $H \overset{p^{-1},m^*}{\Longrightarrow} G$ *using the* inverse production $p^{-1} : (R \xleftarrow{r} K \xrightarrow{l} L)$ *where* $R \xrightarrow{m^*} H$ *is the co-match of* $G \overset{p,m}{\Longrightarrow} H$*.* □

The single-pushout approach is simpler. Graph morphisms are partial maps, productions are simply morphisms $L \xrightarrow{\sigma} R$, and there is no restriction on $m$. Because of that, derivations may have unintuitive side-effects and hence are not invertible in general. Moreover, the approach allows the removal of nodes in unknown contexts. Put differently, any production that removes a node $n$ will also remove automatically all arcs in $G$ incident to $m(n)$, without the designer having any means to prevent it. We feel that for dynamic architecture reconfiguration it is preferable to allow the designer to control precisely in which situations a component (i.e., node) may be removed in order to avoid dangling connectors. For these reasons, we adopt the double-pushout approach.

# Bibliography


[ADG98] Robert Allen, Rémy Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *Fundamental Approaches to Software Engineering*, volume 1382 of *LNCS*, pages 21–37. Springer-Verlag, 1998.

[AGI98] Robert J. Allen, David Garlan, and James Ivers. Formal modelling and analysis of the HLA component integration standard. *Software Enginnering Notes*, 23(6):70–79, November 1998. Proceedings of the Sixth International Symposium on the Foundations of Software Enginnering.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.

[BCM88] Jean-Pierre Banâtre, A. Coutant, and Daniel Le Métayer. A parallel machine for multiset transformation and its programming style. *Future Generation Systems*, pages 133–144, 1988.

[BKK+96] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. In *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.

[BM96] Jean-Pierre Banâtre and Daniel Le Métayer. Gamma and the chemical reaction model: Ten years after. In Jean-Marc Andreoli, Chris Hankin, and Daniel Le Métayer, editors, *Coordination programming: mechanisms, models and semantics*, pages 3–41. Imperial College Press, 1996.

[CDS96] *Proceedings of the Third International Conference on Configurable Distributed Systems*. IEEE Computer Society Press, 1996.

[CDS98] *Proceedings of the Fourth International Conference on Configurable Distributed Systems*. IEEE Computer Society Press, 1998.

[CELM96] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89. Elsevier, 1996.

[CFM98] Paolo Ciancarini, F. Franzè, and Cecilia Mascolo. A coordination model to specify systems including mobile agents. In *Proceedings of the Ninth International Workshop on Software Specification and Design*, pages 96–105. IEEE Computer Society Press, 1998.

[CI99] Flavio Corradini and Paola Inverardi. Model checking of CHAM descriptions of software architectures. Position paper for WICSA1, February 1999.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design—A Foundation*. Addison-Wesley, 1988.

[CMR+96a] Andrea Corradini, Ugo Montanari, F. Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation, part I: Basic concepts and double pushout approach. Technical Report TR-96-17, University of Pisa, March 1996.

[CMR96b] Andrea Corradini, Ugo Montanari, and Francesca Rossi. Graph processes. *Fundamentae Informatica*, 26(3–4):241–266, 1996.

[CPT99] Carlos Canal, Ernesto Pimentel, and José M. Troya. Specification and refinement of dynamic software architectures. In *Software Architecture*, pages 107–125. Kluwer Academic Publishers, 1999.

[Cre91] C. Creveuil. *Techniques d'analyse et de mise en œuvre des programmes Gamma*. PhD thesis, University of Rennes, 1991.

[edc97] Description of EDCS technology clusters. *ACM SIGSOFT Software Engineering Notes*, 22(5):33–42, September 1997.

[EHK+96] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation part II: Single pushout approach and comparison with double pushout approach. Bericht Nr. 96-20, Technische Universität Berlin, Fachbereich 13, Informatik, 1996.

[EM85] Hartmut Ehrig and G. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*. Springer-Verlag, 1985.

[End94] Markus Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In *Proceedings of the 12th Brazilian Symposium on Computer Networks*, pages 175–187, 1994.

[FF96] Nissim Francez and Ira Forman. *Interacting Processes*. Addison-Wesley, 1996.

[Fia96] José Luiz Fiadeiro. On the emergence of properties in component-based systems. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 421–443. Springer-Verlag, 1996.

[FL97] José Luiz Fiadeiro and Antónia Lopes. Semantics of architectural connectors. In *Proceedings of TAPSOFT'97*, volume 1214 of *LNCS*, pages 505–519. Springer-Verlag, 1997.

[FM94] José Luiz Fiadeiro and Tom Maibaum. Category theory for the object technologist. Slides for a OOPSLA tutorial, 1994.

[FM95] José Luiz Fiadeiro and Tom Maibaum. Interconnecting formalisms: Supporting modularity, reuse and incrementality. In *SIGSOFT'95: Third Symposium on Foundations of Software Engineering*, pages 72–80. ACM Press, 1995.

[FM96] José Luiz Fiadeiro and Tom Maibaum. A mathematical toolbox for the software architect. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 46–55. IEEE Computer Society Press, 1996.

[FM97] José Luiz Fiadeiro and Tom Maibaum. Categorial semantics of parallel program design. *Science of Computer Programming*, 28:111–138, 1997.

[FWM99] José Luiz Fiadeiro, Michel Wermelinger, and José Meseguer. Semantics of transient connectors in rewriting logic. Position Paper for the First Working International Conference on Software Architecture, February 1999.

[GAK99] George Yanbing Guo, Joanne M. Atlee, and Rick Kazman. A software architecture reconstruction method. In *Software Architecture*, pages 15–33. Kluwer Academic Publishers, 1999.

[Gar98] David Garlan. Higher-order connectors. Position paper for the Workshop on Compositional Software Architectures, January 1998.

[GK96] Kaveh Moazami Goudarzi and Jeff Kramer. Maintaining node consistency in the face of dynamic change. In CDS96 [CDS96], pages 62–69.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. ACME: An architecture description interchange language. In *Proceedings of the IBM Center for Advanced Studies Conference*, pages 169–183, November 1997.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3–4), 1996.

[HIM99] Dan Hirsch, Paola Inverardi, and Ugo Montanari. Modelling software architectures and styles with graph grammars and constraint solving. In *Software Architecture*, pages 127–143. Kluwer Academic Publishers, 1999.

[HP93] Christine Hofmeister and James Purtilo. Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 101–110, Pittsburgh, May 1993. IEEE Computer Society Press.

[IW95] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[IWY97] Paola Inverardi, Alexander L. Wolf, and Daniel Yankelevich. Checking assumptions in component dynamics at the architecture level. In *Coordination Languages and Models*, volume 1282 of *LNCS*, pages 46–63. Springer-Verlag, 1997.

[IWY98] Paola Inverardi, Alexander L. Wolf, and Daniel Yankelevich. Behavioral type checking of architectural components based on assumptions. Technical Report CU-CS-861-98, Department of Computer Science, University of Colorado, April 1998.

[IY96] Paola Inverardi and Daniel Yankelevich. Relating CHAM descriptions of software architectures. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 66–74. IEEE Computer Society Press, 1996.

[Kat93] S. Katz. A superimposition control construct for distributed systems. *ACM TOPLAS*, 15(2):337–356, 1993.

[Kin93] T. Kindberg. Reconfiguring client-server systems. Technical Report QMW-DCS-1993-630, Queen Mary and Westfield College, Department of Computer Science, March 1993.

[KM85] Jeff Kramer and Jeff Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, 11(14):424–435, April 1985.

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[KM98] Jeff Kramer and Jeff Magee. Analysing dynamic change in distributed software architectures. *IEE Proceedings—Software*, 145(5):146–154, October 1998.

[LF99] Antónia Lopes and José Luiz Fiadeiro. Using explicit state do describe architectures. In *Proceedings of Fundamental Approaches to Software Engineering*, number 1577 in LNCS, pages 144–160. Springer-Verlag, 1999.

[Lop99] Antónia Lopes. *Não-determinismo e Composicionalidade na Especificação de Sistemas Reactivos*. PhD thesis, Universidade de Lisboa, January 1999.

[LV95] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[Mar96] Alfio Martini. Elements of basic category theory. Bericht Nr. 96-5, Technische Universität Berlin, Fachbereich 13, Informatik, 1996.

[Mas99] Cecilia Mascolo. MobiS: A specification language for mobile systems. In *Proceedings of the Third International Conference on Coordination Languages and Models*, volume 1594 of *LNCS*, pages 37–52. Springer-Verlag, 1999.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference*, Barcelona, September 1995.

[Med96] Nenad Medvidovic. ADLs and dynamic architecture changes. In *Joint Proceedings of the SIGSOFT'96 Workshops*, pages 24–27. ACM Press, 1996.

[Med97] Nenad Medvidovic. A classification and comparison framework for software architecture description languages. Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, February 1997.

[MEN96] Alfio Martini, Harmut Ehrig, and Daltro Nunes. Graph grammars - an introduction to the double-pushout approach. Bericht Nr. 96-6, Technische Universität Berlin, Fachbereich 13, Informatik, 1996.

[Mes96] José Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proceedings of the 7th International Conference on Concurrency Theory*, volume 1119 of *LNCS*, pages 331–372. Springer-Verlag, 1996.

[Mét98] Daniel Le Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–553, July 1998.

[MG99] Kaveh Moazami-Goudarzi. *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College London, March 1999.

[MGW97] Robert T. Monroe, David Garlan, and David Wile. *Acme StrawManual*, November 1997.

[Mil99] Robin Milner. *Communicating and Mobile Processes: the π-calculus*. Cambridge University Press, 1999.

[MK96a] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 3–14. ACM Press, 1996.

[MK96b] Jeff Magee and Jeff Kramer. Self organising software architectures. In *Joint Proceedings of the SIGSOFT'96 Workshops*, pages 35–38. ACM Press, 1996.

[MKG99] Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. Behaviour analysis of software architectures. In *Software Architecture*, pages 35–50. Kluwer Academic Publishers, 1999.

[MKS89] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing distributed systems in CONIC. *IEEE Transactions on Software Engineering*, 15(6):663–675, June 1989.

[Mon98] Robert T. Monroe. Capturing software architecture design expertise with armani. Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University, October 1998.

[MP91] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[MP98] Jeff N. Magee and Dewayne E. Perry. Welcome to ISAW-3. In *Third International Software Architecture Workshop*, pages vii–viii. ACM Press, 1998.

[MR96] Peter J. McCann and Gruia-Catalin Roman. Mobile UNITY: A language and logic for concurrent mobile systems. Technical Report WUCS-97-01, Department of Computer Science, Washington University in St. Louis, December 1996.

[MR98] Peter J. McCann and Gruia-Catalin Roman. Compositional programming abstractions for mobile computing. *IEEE Transactions on Software Engineering*, 24(2), February 1998.

[MVS85] Tom Maibaum, Paulo Veloso, and M. Sadler. A theory of abstract data types for program development: Bridging the gap? In *TAPSOFT'85*, volume 186 of *LNCS*, pages 214–230. Springer-Verlag, 1985.

[Ore96] Peyman Oreizy. Issues in the runtime modification of software architectures. Technical Report UCI-ICS-TR-96-35, Department of Information and Computer Science, University of California, Irvine, August 1996.

[Ore98] Peyman Oreizy. Issues in modeling and analyzing dynamic software architectures. In Debra Richardson, Paola Inverardi, and Antonia Bertolino, editors, *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis*, pages 54–57, July 1998.

[OT98] Peyman Oreizy and Richard N. Taylor. On the role of software architectures in runtime system reconfiguration. *IEE Proceedings—Software*, 145(5):137–145, October 1998.

[Pei91] Benjamin C. Peirce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.

[Per97] Dewayne E. Perry. State-of-the-art: Software architecture. In *Proceedings of the 19th International Conference on Software Engineering*, pages 590–591. ACM Press, 1997. Slides available from http://www.bell-labs.com/user/dep/work/swa/icse97.vg.ps.gz.

[Plu95] Detlef Plump. On termination of graph rewriting. In *Proceedings of the 21st International Workshop on Graph-Theoretic Concepts in Computer Science*, number 1017 in LNCS, pages 88–100. Springer-Verlag, 1995.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[RMP97] Gruia-Catalin Roman, Peter J. McCann, and Jerome Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. *ACM TOSEM*, 6(3):250–282, July 1997.

[SG94] Mary Shaw and David Garlan. Characteristics of higher-level languages for software architecture. Technical Report CMU-CS-94-210, School of Computer Science, Carnegie Mellon University, December 1994.

[SG96a] Mary Shaw and David Garlan. Formulations and formalisms in software architecture. In *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 307–323. Springer-Verlag, 1996.

[SG96b] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[TGM98] Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation*, 1998.

[VPL99] James Vera, Louis Perrochon, and David C. Luckham. Event-based execution architectures for dynamic software systems. In *Software Architecture*, pages 303–317. Kluwer Academic Publishers, 1999.

[Wer97] Michel Wermelinger. A hierarchic architecture model for dynamic reconfiguration. In *Proceedings of the Second International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 243–254. IEEE Computer Society Press, 1997.

[Wer98a] Michel Wermelinger. A simple description language for dynamic architectures. In *Proceedings of the Third International Software Architecture Workshop*, pages 159–162. ACM Press, 1998.

[Wer98b] Michel Wermelinger. Specification, testing and analysis of (dynamic) software architecture with the chemical abstract machine. In *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis*, pages 13–17, 1998.

[Wer98c] Michel Wermelinger. Towards a chemical model for software architecture reconfiguration. *IEE Proceedings—Software*, 145(5):130–136, October 1998.

[Wer99] Michel Wermelinger. Software architecture evolution and the chemical abstract machine. In *Proceedings of the International Workshop on the Principles of Software Evolution*. ACM Press, 1999. To appear.

[WF98a] Michel Wermelinger and José Luiz Fiadeiro. Connectors for mobile programs. *IEEE Transactions on Software Engineering*, 24(5):331–341, May 1998.

[WF98b] Michel Wermelinger and José Luiz Fiadeiro. Towards an algebra of architectural connectors: a case study on synchronization for mobility. In *Proceedings of the Ninth International Workshop on Software Specification and Design*, pages 135–142. IEEE Computer Society Press, 1998.

[WF99] Michel Wermelinger and José Luiz Fiadeiro. Algebraic software architecture reconfiguration. In *Software Engineering—ESEC/FSE'99*, volume 1687 of *LNCS*, pages 393–409. Springer-Verlag, 1999.

[WNF99] Michel Wermelinger, Helder Neto, and João Feliciano. *The* CommUnity *Workbench Version 0.1 User Manual*. Laboratório de Modelos e Arquitecturas Computacionais, July 1999. Available from `http://ctp.di.fct.unl.pt/~mw/sw/cw`.

[Wol97] Alexander L. Wolf. Succeedings of the Second International Software Architecture Workshop. *ACM SIGSOFT Software Engineering Notes*, 22(1):42–56, January 1997.

[Wol98] Dietmar Wolz. *Colimit Library for Graph Transformations and Algebraic Development Techniques*. PhD thesis, Technische Universität Berlin, 1998.

[YM92] A. J. Young and J. N. Magee. A flexible approach to evolution of reconfigurable systems. In *Proceedings of the First International Workshop on Configurable Distributed Systems*, pages 152–163. IEE, 1992.